AN OBJECT ORIENTED GRAPHICAL USER INTERFACE FOR A BANKING SYSTEM

Dissertation submitted to the Jawaharlal Nehru University
in partial fulfilment of the requirements
for the award of the Degree of
MASTER OF TECHNOLOGY

in
COMPUTER SCIENCE

by
G. VENUGOPAL

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110067
INDIA

CERTIFICATE

This is to certify that the thesis entitled "**An object oriented Graphical User Interface for a Banking System**" being submitted by me to Jawaharlal Nehru University in partial fulfilment of the requirements for the award of the degree of Master of Technology, is a record of original work done by me under the supervision of Prof. K. K. Nambiar, Professor, School of Computer & Systems Sciences, Jawaharlal Nehru University, during the Monsoon Semester 1993.

The results reported in this thesis have not submitted in part or full to any other University or Institution for the award of any degree etc.

G. Venugopal

G. VENUGOPAL

4-1-94

Prof. K. K. Bharadwaj
Dean, SC & SS
J.N.U, New Delhi
Pin 110067

K K Nambiar

Prof. K. K.Nambiar
Professor, SC & SS
J.N.U, New Delhi
Pin 110067

ACKNOWLEDGEMENTS

I wish to acknowledge with a deep sense of gratitude the guidance and inspiration offered by my supervisor Prof. K. K. Nambiar. His knowledge, analytical thinking power and erudite scholastic experience led me to the completion of the thesis with a sense of satisfaction and accomplishment. It would have been impossible for me to come out successfully without his constant guidance and encouragement.

I extend my thanks to Prof. R. G. Gupta and Prof K. K. Bharadwaj for providing me the opportunity to undertake the project. I would also like to thank the authorities of our school for providing me the necessary facilities to complete my project.

I take this oppotunity to thank my brothers Dr. G. V. N. Appa Rao and Dr. G. Hanumanta Rao for giving me the moral boost and support during the hour of crisis. I would also like to thank all my friends and colleagues, in particular Pramod, Sujit and Venu for helping me during the crucial moments of the project.

CONTENTS

Chapter 1

Introduction

The development of a small prototype of an object oriented Graphical User Interface (GUI) for a banking system is discussed in detail in this thesis. Computers have given us the power to think in a graphic and intuitive fashion. As the state of computing matured, users wanted programs that were visually appealing and simple to use. Today we feel the consequences of this around us -- any program that does not have a GUI looks archaic.

The primary means of interaction with computers until recently has been through command-based interfaces. User gives a command, the system responds. But users have to learn a large set of commands to get their job done. In some systems, meaningful terms were used for command names to help the end user. But in other systems the end user had to memorize arcane sequences of key strokes to accomplish certain tasks. The **mouse** as a pointing device has simplified activities such as procedure and command invocation and data movement within the system.

Graphical User Interfaces are systems that allow creation and manipulation of user interfaces employing **windows, icons, dialog boxes, menus, list boxes, bitmaps, buttons, check boxes, radio buttons, group boxes, combo boxes, edit controls, scroll bars**, mouse and keyboard event handling. GUIs are the state-of-the-practice interfaces. Some people call them WIMP interface: Windows, Icons, Menus and Pointers. Smalltalk MVC, Apple Macintosh Toolbox, Microsoft Windows, NeXT NeXTStep and X-windows are some examples of GUIs. With the introduction of Microsoft Windows and Presentation Manager, the Microsoft had remarkbly changed the landscape of user interfaces for personal computers. Direct

manupulation of computer using GUIs is more intuitive than traditional command based interfaces. In GUIs the textual data is not the only form of interaction. Icons represent concepts such as file folders, waste baskets and printers. Icons symbolize words and concepts commonly applied in different situations. Users perform their day-to-day work on the computer by simply selecting or moving icons and objects on the screen. The GUIs provide an Application Programming Interface (API) that allows users to:

- Create screen objects
- Draw screen objects
- Monitor mouse activations

Microsoft Windows offers many benefits to both users and developers. Benefits to users include:

- if you know how to use one windows application, you know how to use them all.
- no need to setup devices and drivers for each application. Windows provides drivers to support various vendors peripherals.
- multitasking: the ability to run many applications at once. you can open and use any number of overlapped windows to display information in any number of ways. Windows controls the screen.
- data interchange between different windows applications. You can call a Paint brush picture into a Write document, you can transfer data between your application and Clip board, etc.
- access to more memory. Windows supports protected mode on the 80286, 80386, and i486; it supports virtual memory on the 80386 and i486.

Benefits to developers include:

- device-independent graphics, so graphical applications run on all standard display adapters.
- inherent support for wide range of printers, monitors, and pointing devices such as mice and track balls.
- data interchange between different window applications.
- more memory for large programs.
- support for menus, icons, bitmaps, dialog boxes and more.
- a powerful library of graphics routines.

A window is the primary input and output device of any Microsoft Windows (or simply Windows) application. It is the only access to the system display for the application. A window is a combination of a title bar, a menu bar, scroll bars, borders, and other features that occupy a rectangle on the system display. Although an application has exclusive rights to a window created by it, the management of the window is actually a collabarative effort between the application and Windows. Windows maintains the position and appearance of the window, manages standard window features such as the border, scroll bars, and title, and carries out many tasks initiated by the user that affect the window. The application has complete control over the appearance of its window's client area. Each window of the application must have a corresponding *window function.* The window function receives window messages that it should respond accordingly. Every windows application receives input through an application queue and its chief feature is the ***message loop.*** The message loop retrieves input messages from the application queue and dispatches them to the appropriate windows.

Object orientation comes to the rescue of programmer, by lightening the burden of user-interface development. New interface concepts require the development of complex software to display screen objects and handle their events. Simple tasks such as displaying a featureless window require several pages of code in a high level language such as C. Most of the GUIs present an object-oriented layer on top of Application Programming Interface (API) to simplify design and the development of user interfaces. Examples of these layers are Borland's ObjectWindows Library (OWL) for Microsoft Windows, MacApp for Macintosh Toolbox and WhitWater Group's Actor for Microsoft Windows.

The prototype of the banking system has been implemented in Microsoft Windows using Borland's ObjectWindows. ObjectWindows eases Windows application development by providing:

- a consistent, intuitive and simplified interface to Windows.
- supplied behaviour for window management and message processing.
- a basic framework for structuring a Windows application.

In this project we have taken the banking example as a small prototype of a real world banking system. The facilities provided in our Savings Bank system are:

- Opening of a new account for a customer.
- Depositing money to a customer's account.
- Withdrawing money from a customer's account.
- Transferring money from one account to another account.
- Viewing a customer's account details.

- Modifying a customer's address
- Opening of a new branch for the bank.
- Calculation of interest for each account of the bank.
- Viewing a branch's details .

Chapter 2 introduces the basic concepts of object orientation and principles of writing ObjectWindows applications. It gives details on the Borland's ObjectWindows Class library. It also gives the structure and memory requirements of an ObjectWindows program. Chapter 3 gives the design and implementation of the prototype of the GUI for the banking system developed. Chapter 4 gives a sample session with the program. Chapter 5 discusses the future improvements that can be made to this prototype. Chapter 6 gives a partial listing of the program. References used in the project are given at the end of the thesis. Also a glossary has been provided for the technical terms used in the thesis.

Lack of time prevented us from implementing a full fledged savings bank system. The future improvements that can be made to this prototype are given in the Conclusions at the end of the thesis.

Chapter 2

Research Methodology

2.1 Object Oriented Concepts

Object Oriented Concepts offer the potential for significant improvements in the software development process. The object oriented approach builds on the strengths of traditional technologies (i.e., cohesion, coupling, modularity and simplicity) and emphasizes on data abstraction, encapsulation, information hiding, inheritance, polymorphism and reuse. Object orientation is based on encapsulating code and data into a single unit, called an Object.

2.1.1 Object

Object is a well defined **abstraction** of a real world entity. Objects are capsules of behaviour (functions) and state (data), whose internals are hidden from other objects that use their services. Objects can be as small and simple as character strings and icons and as large and complex as databases and servers. In general an object has associated with it:

- a set of **instance variables** that contain the data for the object.
- a set of **messages** to which the object responds. Message is a call to a procedure. In an object oriented interface a message is issued when the user gestures by clicking on a Ok button, for example.
- a **method** is a procedure that performs services. Typically, an object has one method for each operation or message it supports. Methods are also called member functions.

An object implementation defines the format of data associated with an object as well as how the methods are to manipulate that data. Multiple objects may share parts of an implementation. Although the objects share executable code, each object typically has its own copy of the data (some data might be shared). Since the only way that an external object can interact a chosen object is through that object's public interface, i.e., the set of externally known messages to which that chosen object responds, it is possible to modify the methods and variables without affecting other objects. This property of hiding the internal implementation of an object from the external system by binding together both data and messages of the object is called **Encapsulation** or **Information hiding.** Encapsulation protects objects from inadvertent modification of their data members. To modify a data member of the object, the user of the object must explicitly invoke a method that modifies the data member of the object.

A service or a message can have different implementations for different objects, which can produce observably different behaviour, although there is a common intent. A user can issue requests for the service (the requests identify a common operation); an appropriate implementation is selected for each request. The concept of an operation with mutiple implementations is called **Overloading**. The selection of code to perform a service, called binding , is based on the objects identified in the request. In object oriented programming binding is done at runtime, not at compiletime. In general an object is identified when the request is actually issued, so code could

be selected at that time. This is called **Dynamic binding.** Dynamic binding is also called *Late binding*. Code selection is sometimes based on factors that are known before execution, so code can be selected during program compilation or linking. This is called **Static binding.** Static binding is also called *Early binding.*

2.1.2 Class

Class is a set or collection of objects having common features. Object is an instance of its class. Each object in the class share a common definition, though they differ in the values assigned to instance variables. Examples of classes in our user interface are BtreeEntry, OwnDate, TNewDialog etc. The concept of classes is similiar to Abstract Data Types (ADTs). we treat each class is itself an object and that object includes a data member containing the set of all instances of the class and implementation of a method for the message *new,* which creates a new instance of the class. The advantages of class definitions are:

- Allow better conceptualization of the real world and enhances understandbility. Categorize objects based on common structure and behaviour.
- Enhance the robustness of the system by providing strong type checking and thereby enhance the correctness of the programs.
- Seperate the implementation from the specification, allow the modification and enhancement of the implementation without affecting the public interface. Maintenance of the object oriented applications becomes easy due to the reusability of the classes.

- **example**:

```
structure {
  int x;
  int y;
 } point;

class Circle {
   int     radius;                 //    instance variable   radius
   point  center;                  //    instance variable   center

  Circle ( )                       //    constructor
  {
   radius = 1;                     //    initializing the radius to 1 unit.
   center.x = 0;                   //
   center.y = 0;                   //    initializing the center to origin (0,0)
  }

  void expand( int factor)         //     method   expand
  {
    radius = radius * factor;
   }

  void move( point dispacement)  //    method   move
  {
    center. x =  center.x + dispacement.x;
    center. y =  center.y + dispacement.y;
   }
}

void main ( )
{
    Circle circle1;                //    circle1 is an object of Circle
    circle1. expand(2);            //    we are making radius to 2 units
}
```

The definition of a class **Circle** in **C++** is as shown in the example. The instance variables of class **Circle** are *radius* and *center*. The member functions or methods of class **Circle** are *expand* and *move* which change the instance variables *radius* and *center*. **You** cannot modify/access the instance variables *radius* and *center* without using the member functions *expand* and *move* from outside the class. The member function *Circle* whose

name is same as that of the class name **Circle** is called **constructor** of the class **Circle**. When an object of a class is created, constructor is automatically executed. Constructors are mainly used for initialization of instance variables.

2.1.3 Inheritance

Inheritance is a mechanism for sharing the code or behaviour common to a collection of classes. It factors shared properties of classes into base classes and reuse them in the definition of derived classes without modifying already specified classes. Base classes and derived classes are called superclasses and subclasses respectively. We can therefore specify incremental changes of class behaviour in derived classes without modifying already specified classes. Derived classes inherit the code and data of base classes, to which they can add new data members and new member functions. The derived classes can also refine the definition of the base class by replacing or refining the individual methods or member functions. You could design a graphics - object class, for example to define common behaviour for graphics objects with the expectation that the definition of *draw* member function would be replaced in each inheriting class that defines the specific graphics object.

Inheritance plays an important role in modelling of a system because it can express relations among behaviours such as classification, specialization, generalization, approximation and evolution. Inheritance gives the effect of copying and editing the textual definition of a class to produce a new definition, except that changes to old definition eventually propagate

to the new definition. Single Inheritance permits the incremental changes from only one base class where as Multiple Inheritance supports the incremental evolution of artifacts from several base classes.

- **examples**

A class called **titled - window** could be defined to inherit from the class **window.** The **titled - window** class would add a definition for the *title* data member and member functions to implement the operations *get-title* (to return the title) and *set-title* (to change the title).

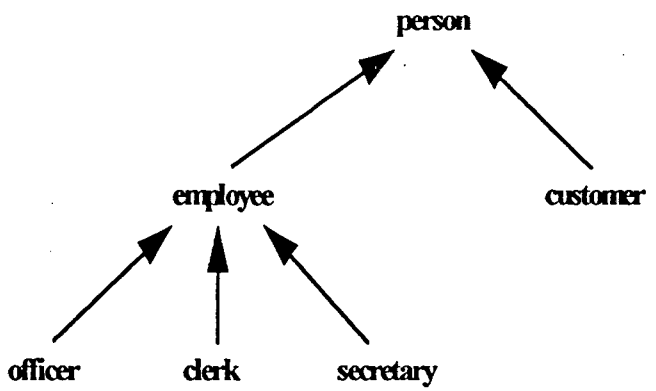

Fig 2.1 Class hierarchy for a tax office example

Figure 2.1 shows the class hierarchy of a tax office example. Both employees and customers are persons. So variables and methods like *name, address, change address* etc that apply both to employees and customers are associated with the **person** class. Variables and methods specific to employees like *salary*, *Issue salary* etc are associated with the **employee** class. similiarly variables and methods specific to customers like *work-phone-*

number, Is tax Paid are associated with the **customer** class. **Person** is the superclass or base class for both **customer** and **employee.** **Customer** and **employee** are derived classes or subclasses of **person.** **Customer** and **employee** inherit the common properties from **person** class. Customers *specialize* the properties of persons, and persons conversely *generalize* the properties of customers. An object representing an officer contain all the variables of classes **officer, employee** and **person.** Here **officer** is a derived class of **employee. Employee** is a base class for **officer, clerk** and **secretary.** The arrows in the class hierarchy diagram or inheritance diagram are from derived classes to base classes. This is because we have to look up to the base class for the properties that are not present in the deived class.

2.2 ObjectWindows concepts

2.2.1 Introduction

ObjectWindows simplifies the development of Windows applications by encapsulating the behaviours that Windows applications commonly perform. ObjectWindows uses the object oriented features of Turbo C^{++} to hide the parts of the Windows API, freeing you from the internals of Windows programming. As a result, you can develop Windows programs with much less time and effort. ObjectWindows provides the following helpful features:

- encapsulation of window information
- abstraction of many Windows API functions
- automatic message response

Many Windows functions require a handle to a window to specify which window they are to act upon, and these functions are usually called from the member functions of a window object. The object holds the handle of its associated window in its HWindow data member. So it can pass HWindow data member as the handle, freeing you from having to specify that item eachtime. The window handle is encapsulated within the object. Like this many of the parameters for Windows functions are stored as data members of interface objects. Thus member functions of window objects can use this data to supply Windows functions with parameters.

ObjectWindows groups related function calls into single member functions that perform higher level tasks. This approach greatly reduces your dependence on the hundreds of Windows API functions, it does not prevent you from calling the API directly.

Windows provides default responses for many of the messages that it sends to an application. When your control, dialog box, or window ignores an incoming message by not defining the corresponding message response member function, Object Windows automatically invokes the default processing supplied by Windows. The appropriate default processing for a control, dialog box, or window is specified by its **DefWndProc** member function.

2.2.2 Structure of an ObjectWindows program

The first requirement of an ObjectWindows application is the definition of an application class derived from base **TApplication** class. By convention the types (Classes and instances of classes) are usually prefixed by the letter **T**, pointers to types by letters **PT** and references to types by letters

RT. An ObjectWindows applictication's main program normally consists of just three statements. The first statement of the **WinMain** (the starting point of a Windows main program) constructs the applicaton object by calling its constructor. The constructor initializes the data members of the application object. The second statement calls the application's **Run** member function. The third statement returns the final status of the application that ObjectWindows stores in *Status* data member. **Run** calls **InitApplication** and **InitInstance** to perform the first-instance and each instance initialization, respectively. **InitMainWindow** is then called to create a main window. **Run** then sets the application in motion by calling **MessageLoop** to begin processing incoming Windows messages, which directly affect the application's flow. **MessageLoop** calls member functions that process particular incoming messages. For example **MessageLoop** calls **WMLButtonDown** member function on receiving a *WM_LButtonDown* message (which occurs when the user clicks the left mouse button). **Message Loop** is exactly that message loop, which continues running until the application closes. The control of an ObjectWindows application by member functions is shown in the figure 2.2.

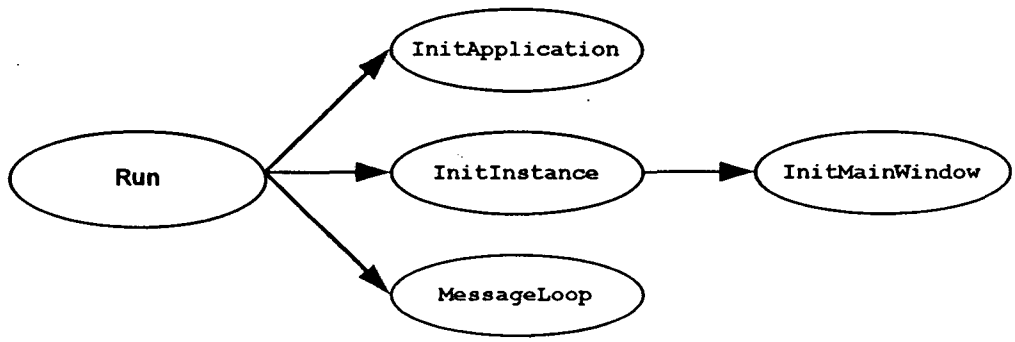

Fig 2.2 Member function calls that control an application's flow

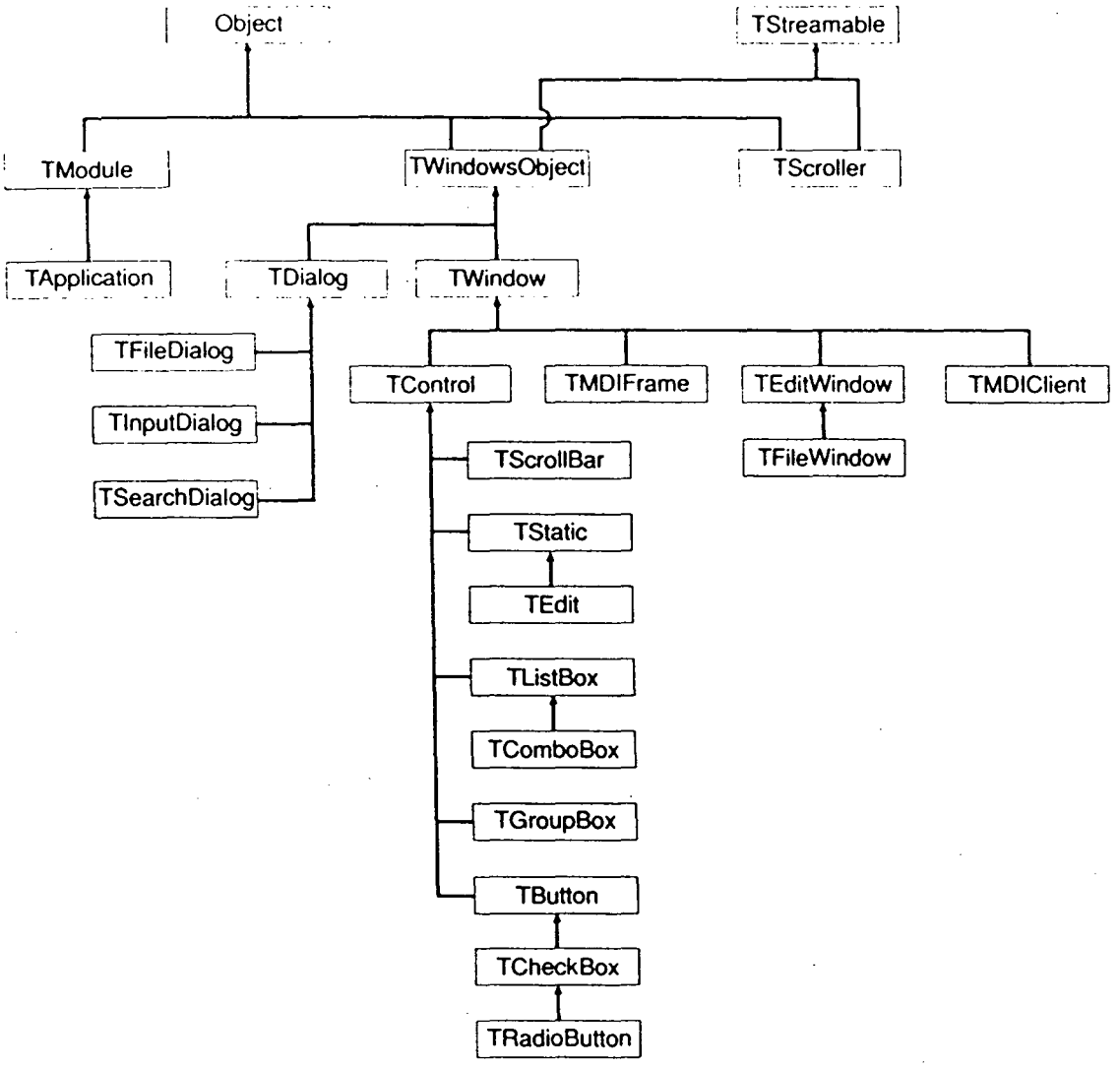

Figure 2.3 ObjectWindows class hierarchy

(ObjectWindows for C++ User's Guide pp 224)

2.2.3 ObjectWindows class hierarchy

ObjectWindows is a comprehensive set of classes that simplifies the development of Windows programs with C^{++}. You can derive new classes from ObjectWindows classes using inheritance. The ObjectWindows class hierarchy is shown in the Figure 2.3.

Object is the base class for all ObjectWindows derived classes. **TApplication** which defines the behaviour of all ObjectWindows applications is derived from **TModule** which itself is derived from **Object**. **TModule** provide support for window memory management and error - processing. **TWindowsObject** is an abstract class, derived from **Object** and **Tstreamable**, that defines the fundamental behaviour shared by all ObjectWindows **interface objects**. Objects representing windows, dialog boxes and controls are called user interface objects or simply interface objects. **TWindowsObject** provides member functions to handle creation, message processing and destruction of window objects.

TWindow is a general purpose window class which can represent main, **pop-up**, or **child windows** of an application. Usually an ObjectWindows application's main window class is derived from **TWindow**. **TDialog** serves as a base class for derived classes that manage Windows dialog boxes. **Dialog objects** serve to facilitate interactive groups of controls such as buttons, list boxes and scroll bars. They are associated with dialog **resources**, They can be run as **modal** or **modeless** dialog boxes. Member functions are provided to handle communication between a dialog and its controls. **TEditWindow** defines a class that allows text editing in a window. **TFileWindow,** derived from **TEditWindow,** defines a class that allows loading and saving text files in addition to text editing in a window. **TFileDialog** defines a dialog that allows the user to choose a file for any purpose, such as opening, editing or saving.

TInputDialog, derived from **TDialog**, defines a dialog box for user input of a single data item.

Control objects such as list boxes, buttons and edit controls, provide a simple means of handling with different kinds of controls defined by Windows. **TContol** is the base class for all control objects and it defines member functions to handle creation and message processing for all control objects. **TButton**, **TCheckBox**, **TRadioButton**, **TListBox**, **TComboBox**, **TStatic**, **TGroupBox** are derived from **TControl. TButton** class represents **push button** interface element in windows. **TCheckBox** and **TRadioButton** classes handle creation and state management of check boxes and radio buttons respectively. **TListBox** handles creation of and selection from Windows list boxes and gives member functions to manipulate items in a list. **TGroupBox** provides member functions to handle a group of selection boxes (check boxes and radio buttons) or other controls. **TStatic** provides member functions that set, query and clear the text of a **static control**. **TEdit** is derived from **TStatic** and it provides extensive text processing capabilities for a windows edit control.

Chapter 3

Design and Implementation

3.1 Hardware and Software Requirements

The Banking system application has been implemented in MS Windows environment using Borland C++ for Windows. The basic hardware requirements for the Banking System application are the same as those of an ObjectWindows application:

- a hard disk
- 2MB of memory or more
- Windows - compatiable graphics display
- Windows 3.0 or later in 386 enhanced mode

3.2 Product Activity

The services provided in our Savings Bank system are:

- Opening of a new account for a customer.
- Depositing money to a customer's account.
- Withdrawing money from a customer's account.
- Transferring money from one account to another account.
- Viewing a customer's account details.
- Modifying a customer's address
- Opening of a new branch for the bank.
- Calculation of interest for each account of the bank.
- Viewing a branch's details.

The design of the Graphical User Interface for the banking system contains two parts. They are bank database design and user interface design. The user interface developed uses the bank database that has been created for accessing and storing transaction data.

3.3 Bank Database Design

The bank database used for the implementation of the user interface is a relational database with the relations **Customer, Deposit, Branch** and **Transaction.** The attributes of the relations are:

Customer: Name, Street, City, Pin

Deposit: AccountNumber, Name, BranchName, Balance, MinBalance

Branch: BranchName, BranchCity, Assets

Transaction: AccountNumber, Type, Date, Time, BranchName, Amount, Balance

Name, AccountNumber and BranchName are primary keys of reations **Customer, Deposit and Branch**. AccountNumber together with Type, Date and Time form the candidate key for the relation **Transaction**. The functional dependencies assumed for the database design are:

Customer: Name → Street Name → City Name → Pin

Deposit: AccountNumber → Name AccountNumber → BranchName

AccountNumber → Balance AccountNumber → MinBalance

Name → AccountNumber Name → BranchName

Name → Balance Name → MinBalance

Branch: BranchName → BranchCity BranchName → Assets

3.4 Bank Database Implementation

The **Customer** relation has been stored in **CUSTOMER.$$$** file. The **Deposit** relation has been stored in **DEPOSIT.$$$** file. The **Transaction** relation has been stored in **TRANSAC.$$$** file.

We have used Borland Container Class Library's **Btree** class to access the elements of the bank database. A **BtreeEntry** class which is derived from **Sortable** has been created. BtreeEntry Class is for storing two related quantities (Key, Value). Given the Key you can have the Value corresponding to that Key. We are using this association with Btree class for storing and accessing a key and that Key's position in a file. BCustomer, BDeposit, BBranch, BATransac and BBTransac are Btree objects. They have been used for accessing transaction data. BCustomer Btree stores the association of customer name and its file position value in CUSTOMER.$$$ file. BDeposit Btree stores the association of account number and its file position value in DEPOSIT.$$$ file. BBranch Btree stores the association of branch name and its file position value in BRANCH.$$$ file. BATransac Btree stores the association of account number and its file position value in TRANSAC.$$$ file. BBTransac Btree stores the association of branch name and its file position value in TRANSAC.$$$ file. Also we have used OwnDate class derived from Date class so as facilitate simple date manipulations.

3.5 User Interface Design

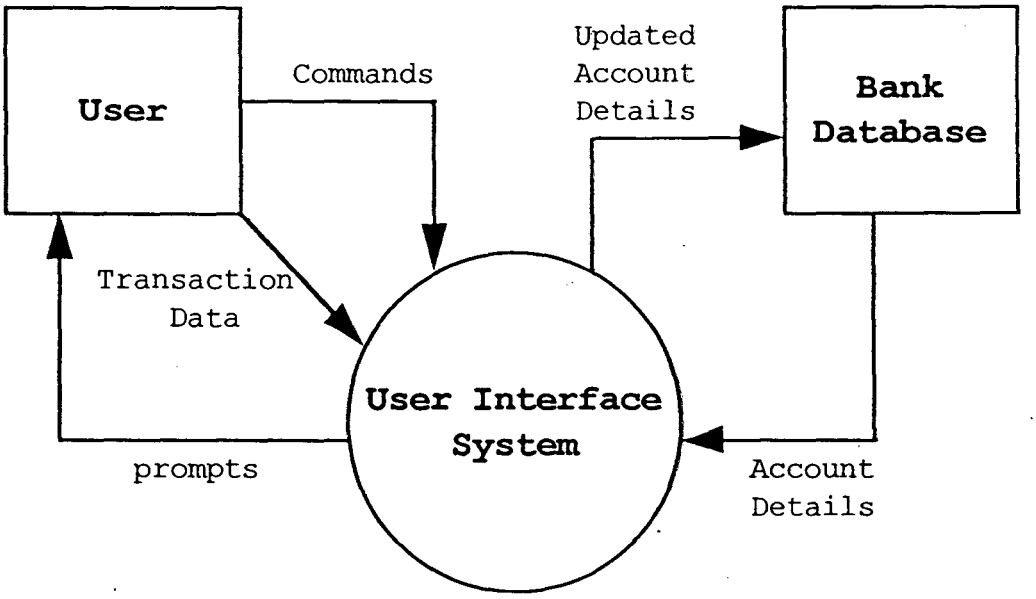

Figure 3.1 Banking System Context Diagram

The user interface system interacts with both the database and the user as shown in the banking system context diagram (Figure 3.1). User interacts with the system by giving commands and transaction data. The system responds to the input given by the user. The system reads account details from the database and writes updated account details to the database.

The psuedo code for the message processing and the application flow is given below. The psuedo code has been given to only those messages the application itself manages. The other messages are processed by Windows and we refer them by **Default Windows Processing.** Care has been

taken that the psuedo code naming conventions resemble the original source code naming conventions for classes, functions and messages. Chapter 6 gives a partial listing of the source code of the application.

3.5.1 Application flow

```
Start
    Construct a TApplication object
    Perform instance initialization
    Construct a TMainWindow object
    repeat
     Process Messages for the application
    until ( message = WM_QUIT )
     Destruct TMainWindow and TApplication objects
     return status
End
```

3.5.1.1 Construct a TApplication object

```
Start
    Initialize TApplication object data members .
End
```

3.5.1.2 Perform instance initialization

```
Start
    Perform first instance initialization
    Perform each instance initialization
    Process InitMainWindow
End
```

3.5.1.3 Constuct TMainWindow object

Start

Load custom control library object

Assign menu to TMainWindow object

Build Customer Btree using customer log file

Build Deposit Btree using deposit log file

Build AccountTransaction Btree using Account transaction log file

Build BranchTransaction Btree using Branch transaction log file

End

3.5.1.4 Process messages for the application

Start

Case Active Object:

TMainWindow ⇒ Process main window messages

TNewDialog ⇒ Process new account dialog messages

TDepositDlg ⇒ Process deposit dialog messages

TWithdrawDlg ⇒ Process withraw dialog messages

T TranferDlg ⇒ Process money transfer dialog messages

TModifyDlg ⇒ Process modify address dialog messages

TViewAccountDlg ⇒ Process customer account details dialog messages

TNewBranchDlg ⇒ Process new branch dialog messages

TInterestDlg ⇒ Process interest rate dialog messages

TBranchDetailsDlg ⇒ Process branch details dialog messages

TInputDialog ⇒ Process TInputDialog messages

End Case

End

3.5.1.4.1 Process main window messages

Start

Case user selection

menuitem New Account	⇒	Execute TNewDialog
menuitem Deposit Money	⇒	Execute TDepositDlg
menuitem Withdraw Money	⇒	Execute TWithdrawDlg
menuitem Transfer Money	⇒	Execute TTransferDlg
menuitem Modify Address	⇒	Process ModifyAddress
menuitem View Account	⇒	Process ViewAccount
menuitem New Branch	⇒	Execute TNewBranchDlg
menuitem Interest	⇒	Process MonthlyUpdation
menuitem Branch Details	⇒	Process BranchDetails
menuitem Exit	⇒	Process CanClose
default	⇒	Default Windows processing

End Case

End

Figure 3.2 shows the message processing for the main window. On receiving a message, the TMainWindow object executes the corresponding member function. For example if the user selects New Account menuitem, the application gets a CM_NEW message from Windows and New member function of the TMainWindow is executed provided the TMainWindow object is active. If TMainWindow object is not active then no action will be taken.

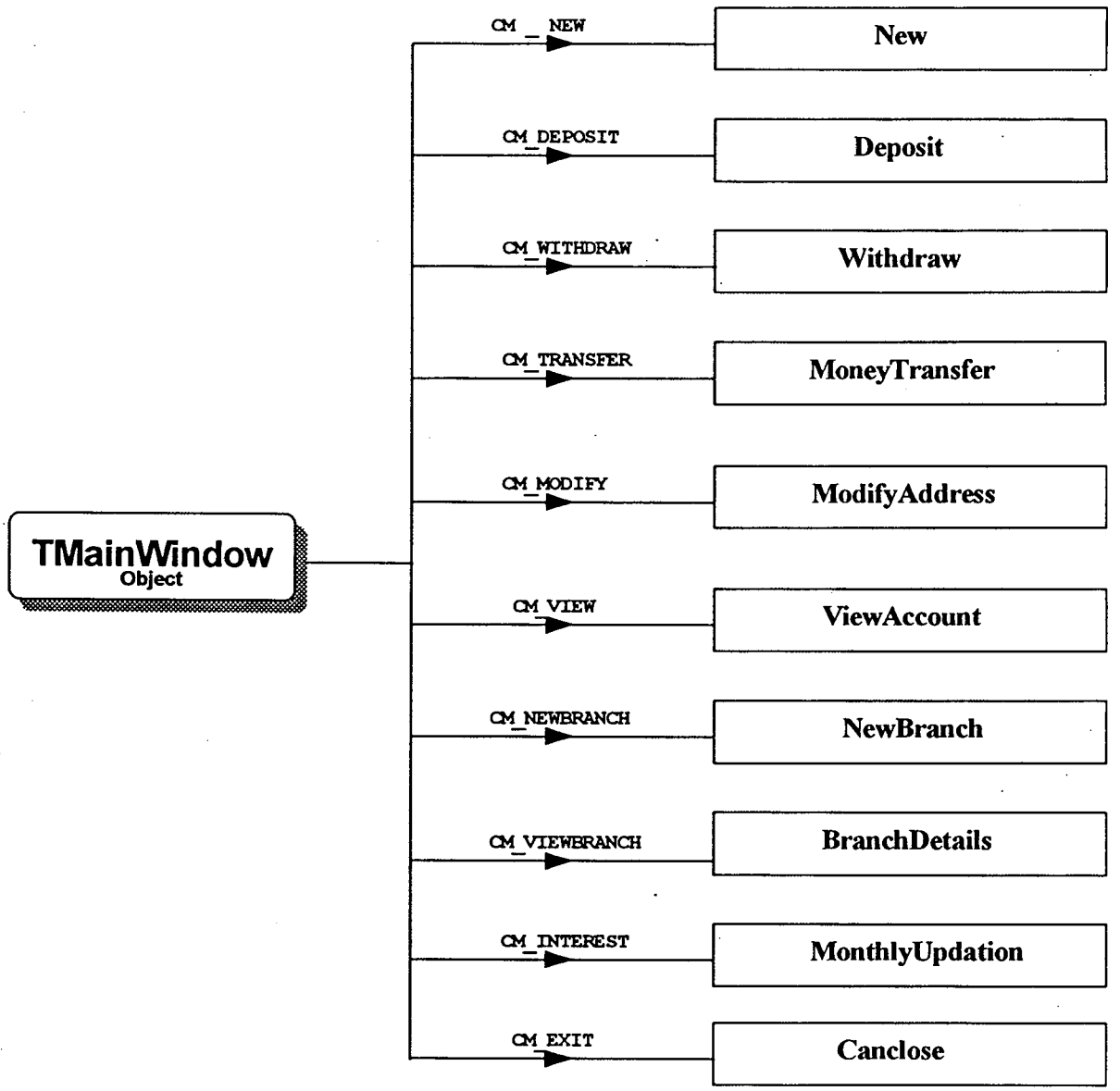

Figure 3.2 Message processing for main window

3.5.1.4.1.1 Execute TNewDialog

```
Start
    Construct TNewDialog's child objects
    Get new AccountNumber for the new account
    Show New Account dialog and Prompt user
    Case selection
    Ok button: If ( Valid Address & Valid InitialDeposit
                    & Valid Branch)
                    Store Address in Customer file
                    Set Minimum Balance to InitialDeposit
                    Store Deposit details in Deposit file
                    Store Transaction details in Transaction file
                    Get BranchAssets from Branch file
                    BranchAssets = BranchAssets + InitailDeposit
                    Store back BranchAssets to Branch file
                    Update all Btree structures execept Branch Btree
                    Update AccountNumber in Bank file
                    Close TNewDialog and return to TMainWindow
               else
                    Give error message
               EndIf
    Cancel button: Default Windows Processing
         default: Default Windows Processing
    End Case
End
```

3.5.1.4.1.2 Execute TDepositDlg

```
Start
        Construct TDepositDlg's child objects
        Show Deposit dialog and prompt user
        Case Selection
        Ok button: If ( Valid AccountNumber & Valid DepositAmount )
                        Get Balance from Deposit file
                        Balance = Balance + DepositAmount
                        Store back Balance in Deposit file
                        Get BranchAssets from Branch file
                        BranchAssets = BranchAssets + DepositAmount
                        Store back BranchAssets to Branch file
                        Store Transaction details in Transaction file
                        Update Transaction Btree structures
                        Close TDepositDlg and return to TMainWindow
                   else
                        Give error message
                    EndIf
        Cancel button:  Default Windows Processing
              default:  Default Windows Processing
        End Case
End
```

3.5.1.4.1.3 Execute TWithdrawDlg

```
Start
        Construct TWithdrawDlg's child objects
        Show Withdraw dialog and prompt user
        Case Selection
        Ok button: If ( Valid Account Number & Valid WithdrawAmount )
                        Get Balance from Deposit file
                        Balance = Balance - WithdrawAmount
                        Store back Balance to Deposit file
                        Get Minimum Balance of the account
                        If ( Balance < Minimum Balance )
                            Set Minimum Balance to Balance
                            Store back Minimum Balance to Deposit file
                        EndIf
                        Get BranchAssets from Branch file
                        BranchAssets = BranchAssets - WithdrawAmount
                        Store back BranchAssets to Branch file
                        Store Transaction details in Transaction file
                        Update Transaction Btree structures
                        Close TWithdrawDlg and return to TMainWindow
                   else
                        Give error message
                    EndIf
        Cancel button:  Default Windows Processing
              default:  Default Windows Processing
        End Case
End
```

3.5.1.4.1.4 Execute TTransferDlg

Start

Construct TTransferDlg's child objects

Show Money Transfer dialog and prompt user

Case Selection

Ok button: If (Valid FromAccountNumber & Valid Transaction Amount & Valid ToAccountNumber)

Get ToBalance from Deposit file

ToBalance = ToBalance + TransactionAmount

Store back ToBalance in Deposit file

Get FromBalance from Deposit file

FromBalance = FromBalance - TransactionAmount

Store back FromBalance to Deposit file

Get Minimum Balance of Fromaccount

If (FromBalance < Minimum Balance)

Set Minimum Balance to FromBalance

Store back Minimum Balance to Deposit file

EndIf

Get FromBranchAssets from Branch file

FromBranchAssets = (FromBranchAssets - TransactionAmount)

Store back FromBranchAssets to Branch file

Get ToBranchAssets from Branch file

ToBranchAssets = ToBranchAssets + TransactionAmount

Store back ToBranchAssets to Branch file

Store Transaction details in Transaction file

Update Transaction Btree structures

Close TTransferDlg and return to TMainWindow

else

Give error message

EndIf

Cancel button: Default Windows Processing

default: Default Windows Processing

End Case

End

3.5.1.4.1.5 Process ModifyAddress

Start

Execute TInputDialog to get AccountNumber input from user

If (Valid AccountNumber)

Get Customer Address from Customer file

Construct TModifyDlg's child objects

Show TModifyDlg and prompt user to change

Case Selection

Ok button: If (Modified)

Store New Address in the Customer file

Update Customer Btree structure

EndIf

Close TModifyDlg and return to TMainWindow

Cancel button: Default Windows Processing

default: Default Windows Processing

End Case

else

Give error message

EndIf

End

3.5.1.4.1.6 Process ViewAccount

Start

Execute TInputDialog to get AccountNumber input from user

If (Valid AccountNumber)

Get Customer Address from Customer file

Get BranchName and Balance from Deposit file

Find the Number of transactions for the account

If (Number of Transactions > 20)

Set DisplayTransactionNumber to 20

else

Set DisplayTransactionNumber to Number of Transactions

EndIf

Get latest DispayTransactionNumber transactions for the account

Construct TViewAccountDlg's child objects

Show TViewAccountDlg with Account Details and prompt user

Case Selection

Ok button: Close TViewAccountDlg and return to TMainWindow

default: Default Windows Processing

End Case

```
        else
            Give   error  message
        EndIf
    End
```

3.5.1.4.1.7 Process MonthlyUpdation

```
    Start
        If ( MonthEnd  &  Interest Not Yet Calculated)
            Execute  TInputDialog to get InterestRate  input from user
            If ( Valid InterestRate )
              Do  for  all  Accounts
                 Get  Minimum  Balance of  Account from  Deposit  file
                 Interest  =  MinBalance  *  InterestRate / (12  *  100 )
                 Balance  =  Balance  +  Interest
                 Set  Minimum Balance  to  Balance
                 Store  back  Balance and Minimum Balance to  Deposit  file
                 Get  BranchAssets  of  Account's Branch from Branch file
                 BranchAssets  =  BranchAssets  +  Interest
                 Store back BranchAssets to  Branch file
                 Store transaction details  in  transaction file
                 Update  Transaction Btree  Structures
              End Do
            else
                  Give  error  message
            EndIf
            Close  TInterestDlg  and  return to  TMainWindow
```

else

Give error message

EndIf

End

3.5.1.4.1.8 Execute TNewBranchDlg

Start

Construct TNewBranchDlg's child objects

Show New Branch dialog and Prompt user

Case selection

Ok button: If (Valid BranchName & Valid BranchCity)

Set BranchAssets to 0

Store BranchName, BranchCity and BranchAssets in Branch file

Close TNewDialog and return to TMainWindow

else

Give error message

EndIf

Cancel button: Default Windows Processing

default: Default Windows Processing

End Case

End

3.5.1.4.1.9 Process BranchDetails

Start

Execute TInputDialog to get BranchName input from user

If (Valid BranchName)

Get BranchCity and BranchAssets from Branch file

Construct TBranchDetailsDlg's child objects

Show TBranchDetailsDlg with Branch Details and prompt user

Case Selection

Ok button: Close TBranchDetailsDlg and return to TMainWindow

default: Default Windows Processing

End Case

else

Give error message

EndIf

End

3.5.1.4.1.10 Process CanClose

Start

Store Customer Btree contents in Customer log file

Store Deposit Btree contents in Deposit log file

Store Branch Btree contents in Branch log file

Store AccountTransaction Btree Contents in AccountTransaction log file

Store BranchTransaction Btree Contents in BranchTransaction log file

End

3.6 Class Inheritance Structure

The Figures 3.3 , 3.4 and 3.5 show the class inheritance structure of the application. The TMainWindow class is derived from TWindow class. TNewDialog, TDepositDlg, TWithdrawDlg, TTransferDlg, TViewAccountDlg, TModifyDlg, TNewBranchDlg and TBranchDetailsDlg classes are derived from TDialog class. TInterestDlg class is derived from TInputDialog class. The OwnDate class is derived from Date class. The BtreeEntry class is derived from Sortable class.

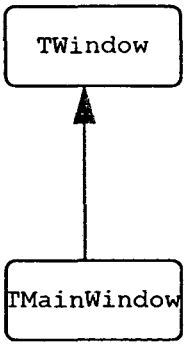

Figure 3. 3 Inheritance Diagram for Application's Main Window

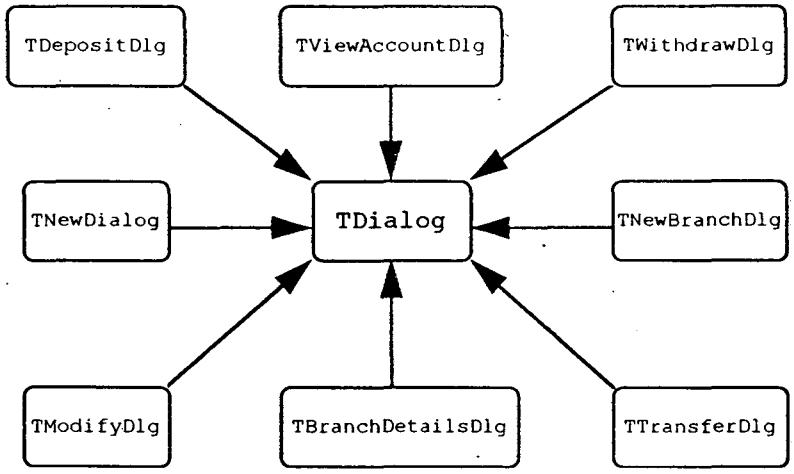

Figure 3. 4 Inheritance Diagram for Windows dialog boxes

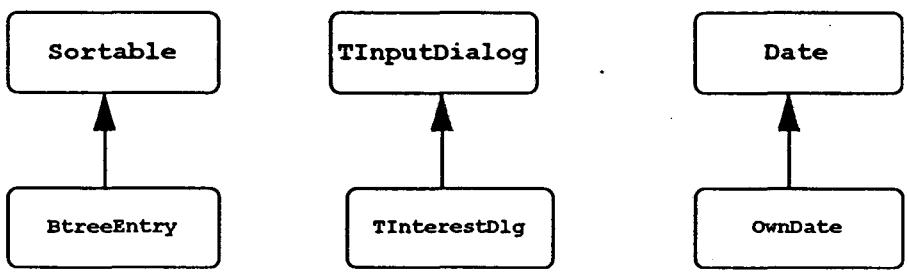

Figure 3.5 Inheritance diagram for BtreeEntry, OwnDate and TInputDialog classes

Chapter 4

Sample Session

The user can either run the Savings Bank application by double clicking the Savings Bank icon or he can do it from Program Manager's File/Run menu item by typing the application's path name.

The application's main window (Figure 4.1) shows the main menu of the application. The **Customer** menu item provides the customer services (such as opening of a new Account, Depositing Money to a customer's account etc). The **Branch** menu item provides the branch services or internal sevices for the bank (such as viewing branch details, interest calculation etc). The user can close the application using **System/Exit.**

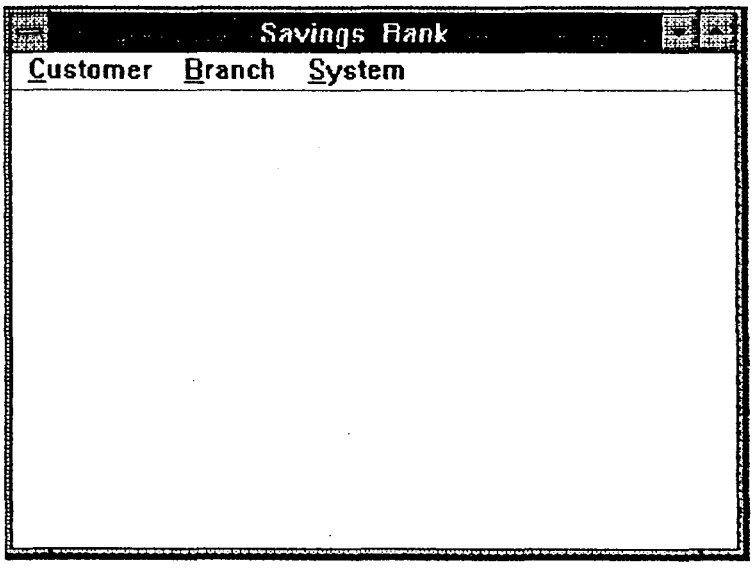

Figure 4.1 The Main Window of the application

The user can see (Figure 4.2) the pulldown menu of the **Customer** menu item by clicking on it or by using Alt + C sequence. He can select one of the services provided by using up/down arrow keys or by pressing the underlined letter of that service or by clicking on that service. Also he can directly select the customer service required by pressing the accelator key of that service from the main window itself.

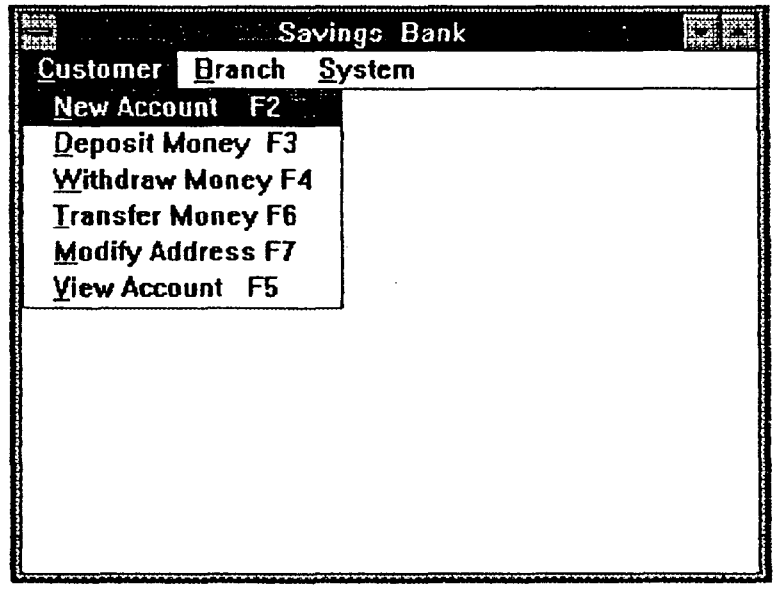

Figure 4.2 The Main Window with the pulldown menu of the Customer menu item

The **Branch** menu item's pulldown menu is as shown in the figure 4.3 which shows the branch services provided by the application to the bank.

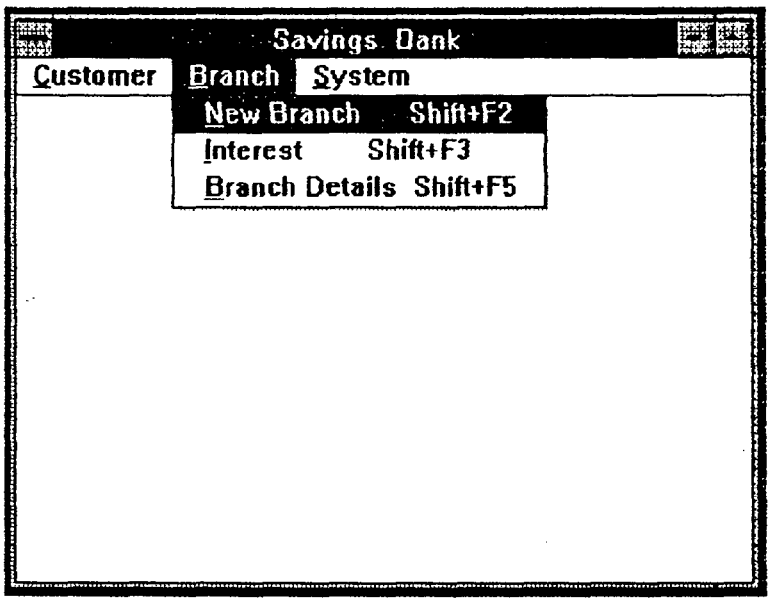

Figure 4.3 The Main Window with the pulldown menu of the Branch menu item

When the user selects the New Account menuitem in the main window, the New Account Dialog shown in Figure 4.4 appears on the screen.

Functionality: Accepts customer's address, the branch name in which customer wants to open his account and initial deposit.

Operation: Choices can be selected by pointer or Tabs

Validation: Checks for erroneous input

Name: Uniqueness of Name

Initial Deposit: Amount should be more than Rs 20.00

Branch: Branch should be present

Account Number: Read only

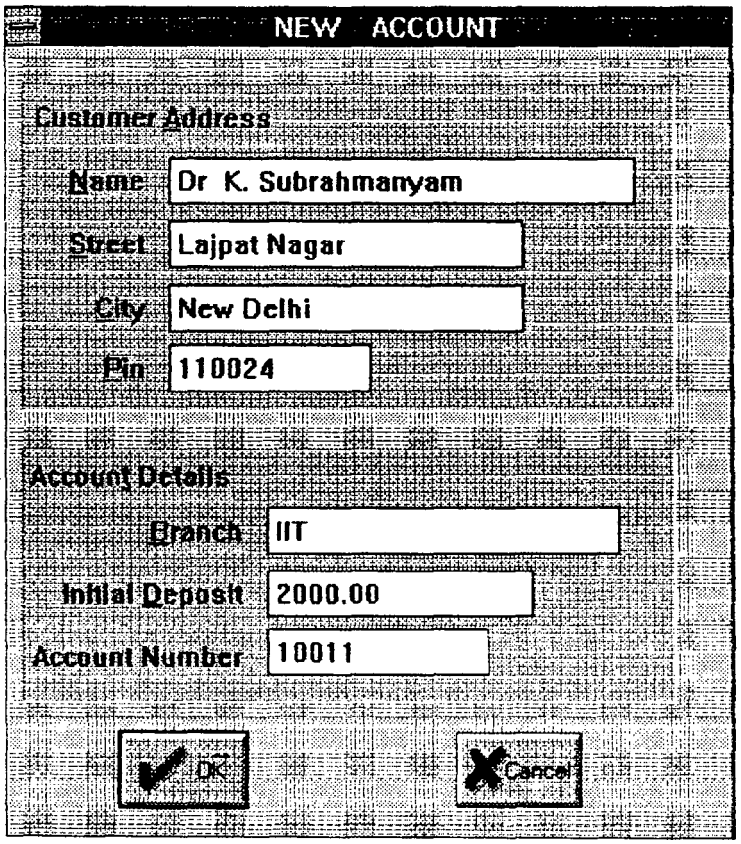

Figure 4.4 New Account Dialog

When the user selects the Deposit Money menuitem in the main window, the Deposit Dialog shown in Figure 4.5 appears on the screen.

Functionality: Accepts customer's Account Number and Deposit Amount

Operation: Choices can be selected by pointer or Tabs

Validation: Checks for erroneous input

Account Number: Account number should be present

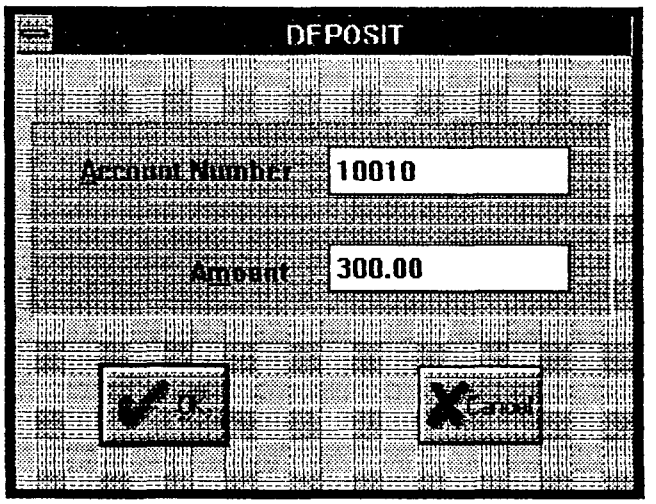

Figure 4.5 Deposit Dialog

When the user selects the Withdraw Money menuitem in the main window, the Withdraw Dialog shown in Figure 4.6 appears on the screen.

Functionality: Accepts customer's Account Number and Withdraw Amount

Operation: Choices can be selected by pointer or Tabs

Validation: Checks for erroneous input

Account Number: Account number should be present

Amount: Sufficient Balance should be present

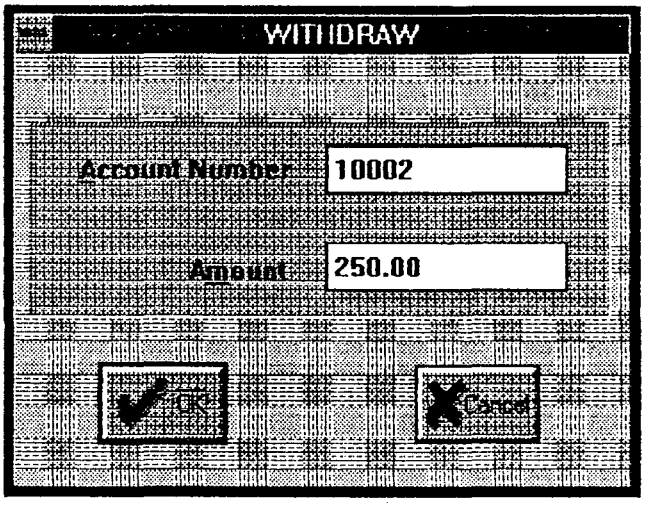

Figure 4.6 Withdraw Dialog

When the user selects the Transfer Money menuitem in the main window, the Money Transfer Dialog shown in Figure 4.7 appears on the screen.

Functionality: Accepts Account Numbers and Transfer Amount

Operation: Choices can be selected by pointer or Tabs

Validation: Checks for erroneous input

Account Number: Account numbers should be present

Amount: Sufficient Balance should be present in From Account

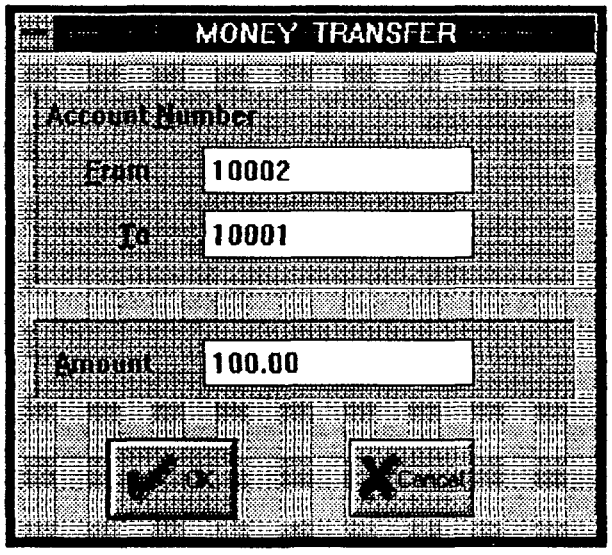

Figure 4.7 Money Transfer Dialog

When the user selects the Modify Address menuitem in the main window, the Account Number Input Dialog shown in Figure 4.8 appears on the screen.

Functionality: Accepts customer's Account Number.

Operation: Choices can be selected by pointer or Tabs

Validation: Checks for erroneous input

Account Number: Account number should be present

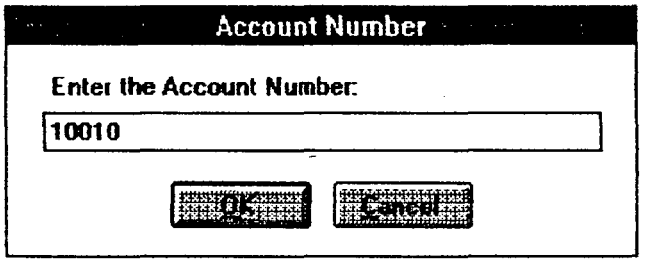

Figure 4. 8 Account Number Input Dialog

If Account Number is valid, the application gets the address of the customer from the database and displays the address to modify on the screen as shown figure 4.9

Functionality: Accept customer's address

Operation: Choices can be selected by pointer or Tabs

Validation: Checks for erroneous input

Name: Uniqueness of name, if name is changed

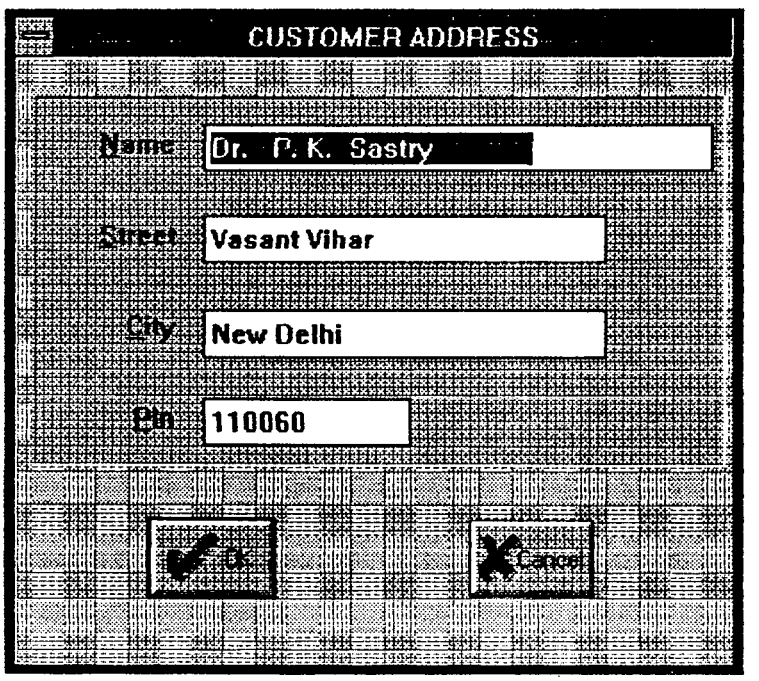

Figure 4.9 Modify Address Dialog

When the user selects the View Account menuitem in the main window the Account Number Input Dialog shown in Figure 4.10 appears on the screen.

Functionality: Accepts customer's Account Number.

Operation: Choices can be selected by pointer or Tabs

Validation: Checks for erroneous input

Account Number: Account number should be present

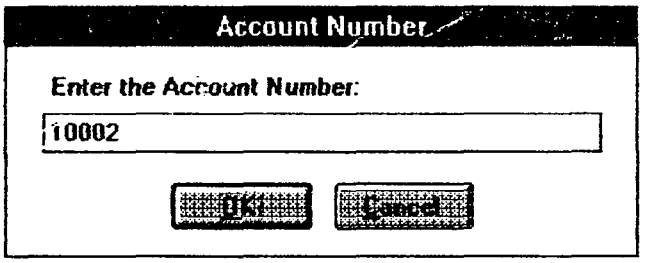

Figure 4.10 Account Number Dialog

If Account Number is valid, the application gets customer's address and account details from the database and displays them on the screen as shown figure 4.11

Functionality: Display Account Details.

Operation: Choices can be selected by pointer or cursor keys and tabs.

Validation: None.

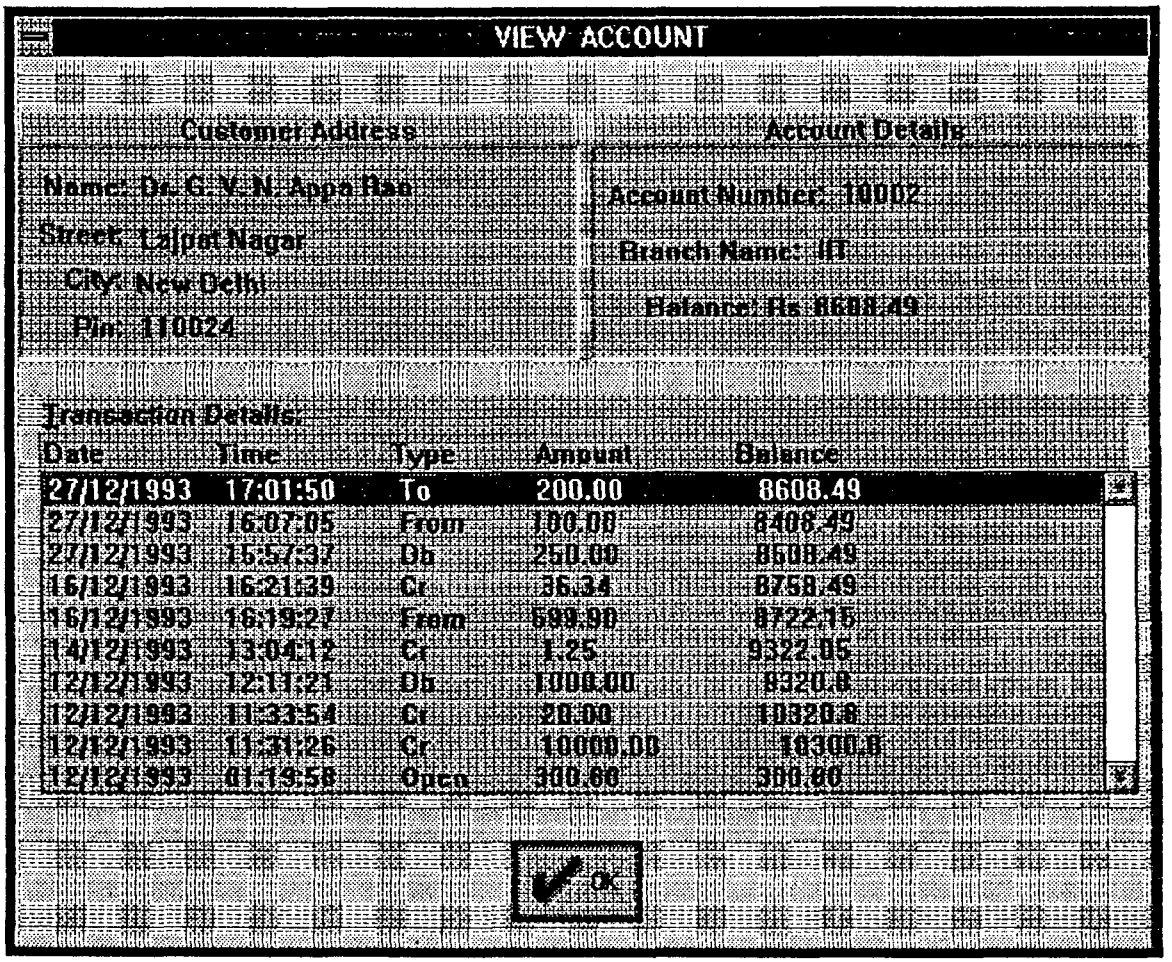

Figure 4.11 View Account Dialog

When the user selects the New Branch menuitem in the main window, the New Branch Dialog shown in Figure 4.12 appears on the screen.

Functionality: Accepts Branch Name and Branch City

Operation: Choices can be selected by pointer or Tabs

Validation: Checks for erroneous input

Branch Name: Uniqueness of Branch Name

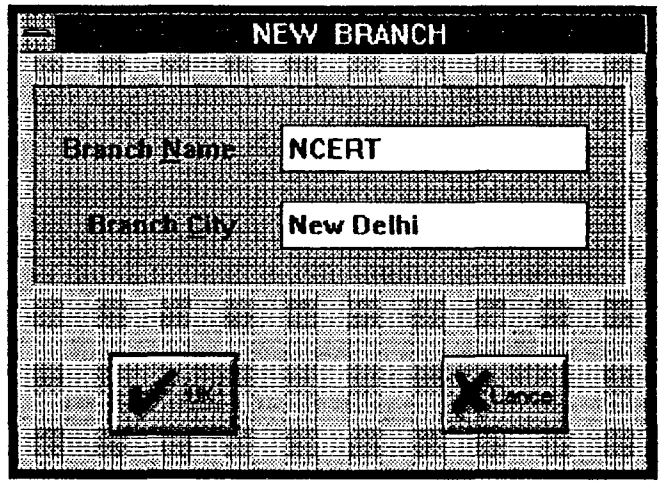

Figure 4.12 New Branch Dialog

When the user selects the Branch Details menuitem in the main window the Branch Name Input Dialog shown in Figure 4.13 appears on the screen.

Functionality: Accepts Branch Name

Operation: Choices can be selected by pointer or Tabs

Validation: Checks for erroneous input

Branch Name: Branch Name should be present

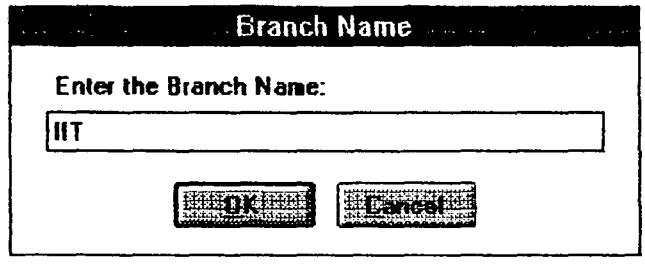

Figure 4.13 Branch Input Dialog

If the Branch Name is valid, the application gets branch details from the database and displays them on the screen as shown figure 4.14

Functionality: Display Branch Details.

Operation: Responds to return key or mouse click on Ok button.

Validation: None.

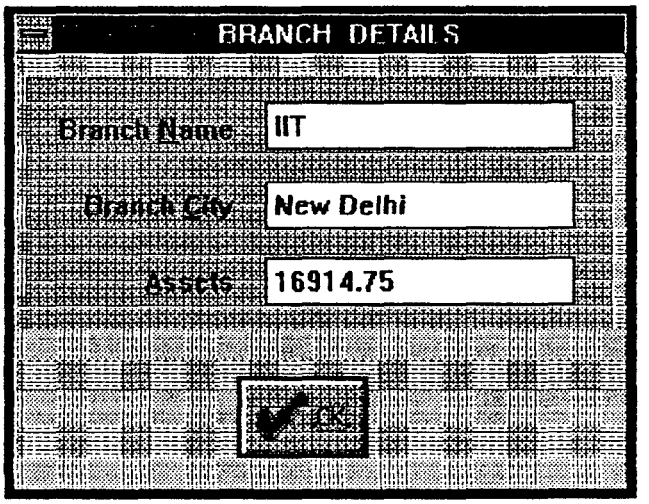

Figure 4.14 Branch Details Dialog

When the user selects the Interest menuitem in the main window, the Interest Rate Input Dialog shown in Figure 4.15 appears on the screen provided the Date is the last day of the month and the Time is past 6 pm.

Functionality: Accepts Interest Rate

Operation: Choices can be selected by pointer or Tabs and default interest rate is 5.0%

Validation: Checks for erroneous input

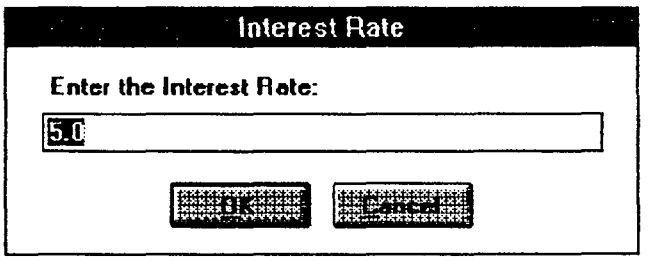

Figure 4.15 Interest Rate Dialog

Chapter 5

Conclusion

The design and implementation of a prototype of an object oriented GUI for a banking system has been studied in this thesis. The power of object orientation makes the design of the prototype straight forward. The improvements that can be made to this application without changing the basic structure of the user interface are many.

The major improvements that can be made to this prototype to make it a real world banking system are:

- Authorization can be provided by giving the users a userID and a password. Security to the bank database can be provided by giving different views for different users thereby restricting the users to access only certain services provided by the bank system.
- The prototype presented supports single user only. It can be changed to a distributed banking system so that multiple transactions can be processed at the same time. This prototype program needs little changes to accomplish this task. This is because the Windows file management functions have been used in this program for accessing the database , which allow data sharing. Shared locks and excusive locks can be put to transaction data. Also Windows allows running multiple instances of the same application at the same time.
- The services provided by this application for the same structure of the database, can be extended with little work. This can be done by adding new message response functions to **TMainWindow** and the code to handle the functions.

- A full fledged help facility can be added to this GUI application.
- The database structure can also be changed with little changes to the GUI application so as to handle a real world banking system.

Lack of time prevented us from simplifying even further the user interface for the banking system.

Chapter 6

Program Listing

The project file of the application contains files ***main.cpp, new.cpp, newb.cpp, deposit.cpp, withdraw.cpp, transfer.cpp, month.cpp, modify.cpp, view.cpp, bdetails.cpp, owndate.cpp, project.rc*** and ***standard.def.*** The header files of the application are ***project.h, owndate.h*** and ***wconst.h.*** we are giving a listing of the main program file ***main.cpp*** , the project's header file ***project.h*** and module definition file ***standard.def*** which will give a brief outline of the application.

The ***main.cpp*** file contains WinMain and member functions for TMainWindow and TMainApp classes. The ***project.h*** file contains class declarations for all window classes and BtreeEntry class. The files ***new.cpp, newb.cpp, deposit.cpp, withdraw.cpp, transfer.cpp, month.cpp, modify.cpp, view.cpp, bdetails.cpp*** and ***owndate.cpp*** contain member functions for TNewDialog, TNewBranchDlg, TDepositDlg, TWithdrawDlg, TTransferDlg, TInterestDlg, TModifyDlg, TViewAccountDlg, TBranchDetailsDlg and OwnDate classes respectively. The ***project.rc*** file the resource data for main menu, dialog boxes and accelarators. The ***standard.def*** file contains the memory stack and memory heap requirements of the application.

6.1 Application's Main Program (MAIN.CPP)

```
#include "project.h"        // header file of the application

/*
    BCustomer B-Tree stores the association of customer name and its
 file pointer value in CUSTOMER.$$$ file. BDeposit Btree stores the
 association of account number and its file pointer value in DEPOSIT.$$$
 file. BBranch Btree stores the association of branch name and its
 file pointer value in BRANCH.$$$ file. BATransac Btree stores the
 association of account number and its file pointer value in TRANSAC.$$$
 file. BBTransac Btree stores the association of branch name and its
 file pointer value in TRANSAC.$$$ file.
*/

Btree BCustomer(5), BDeposit(5), BBranch(5), BATransac(5), BBTransac(5);
long GlobalCustomerPos, GlobalDepositPos, GlobalBranchPos;
char srate[6];


/*
    setstring function pads the string s with blanks until the
  length of the string becomes (c-1).
*/
void setstring(char *s, int c)
{
 int len,nblank,i;
 len = strlen(s)+1;
 nblank=c-len;
 for(i=1;i<=nblank;i++)strcat(s," ");
}
```

```
/*---------------------------------------------*/
/* TMainWindow implementations:  */
/*---------------------------------------------*/

/*
  TMainWindow's constructor assigns the application menu, it

   loads the Borland's custom control library. It builds the

   Btrees from  the LOG files.
*/
TMainWindow::TMainWindow(PTWindowsObject Parent,LPSTR ATitle)
                        : TWindow(Parent, ATitle)
{
 OFSTRUCT of;
 int chandle, dhandle, bhandle, athandle, bthandle;
 char spos[POSLEN], Name[MAXNAMELEN], AccountNumber[ACCOUNTLEN],
Branch[MAXBLEN];

 AssignMenu(200);
 BWCCMod = LoadLibrary("BWCC.DLL");


 chandle = OpenFile("customer.log", &of, OF_READ);
 while(!eof(chandle))
 {
  _lread(chandle, Name, MAXNAMELEN);
  _lread(chandle, spos, POSLEN);
  BCustomer.add( *(Association*) new BtreeEntry(Name, spos));
 }
 _lclose(chandle);


 dhandle = OpenFile("deposit.log", &of, OF_READ);
 while(!eof(dhandle))
 {
  _lread(dhandle, AccountNumber, ACCOUNTLEN);
  _lread(dhandle, spos, POSLEN);
  BDeposit.add( *(Association*) new BtreeEntry(AccountNumber, spos));
 }
 _lclose(dhandle);


 bhandle = OpenFile("branch.log", &of, OF_READ);
 while(!eof(bhandle))
 {
  _lread(bhandle, Branch, MAXBLEN);
  _lread(bhandle, spos, POSLEN);
  BBranch.add( *(Association*) new BtreeEntry(Branch, spos));
 }
 _lclose(bhandle);
```

```
 athandle = OpenFile("atransac.log", &of, OF_READ);
 while(!eof(athandle))
 {
  _lread(athandle, AccountNumber, ACCOUNTLEN);
  _lread(athandle, spos, POSLEN);
  BATransac.add( *(Association*) new BtreeEntry(AccountNumber, spos));
 }
 _lclose(athandle);


 bthandle = OpenFile("btransac.log", &of, OF_READ);
 while(!eof(bthandle))
 {
  _lread(bthandle, Branch, MAXBLEN);
  _lread(bthandle, spos, POSLEN);
  BBTransac.add( *(Association*) new BtreeEntry(Branch, spos));
 }
 _lclose(bthandle);

}


// Defining MainWindow class name
LPSTR TMainWindow::GetClassName()
{
 return APPNAME;
}


/*
    Defining WndClass of the application's main window. The main

  window features are inherited from TWindow class except that the

  Icon we use is different.
*/
void TMainWindow::GetWindowClass(WNDCLASS& AWndClass)
{
  TWindow::GetWindowClass(AWndClass);

  AWndClass.hIcon = LoadIcon(GetApplication()->hInstance,APPNAME);
}



/*
    CanClose member function saves the Btree data into LOG files

  just before the application is closed and it returns TRUE.
*/

BOOL TMainWindow::CanClose()
{

  OFSTRUCT of;
  int chandle, dhandle, bhandle, thandle, tahandle;
```

```
int citems, ditems, bitems, titems, taitems;

String& Name(*new String("")), spos(*new String(""));
String& AccountNumber(*new String("")), Branch(*new String(""));

citems = BCustomer.getItemsInContainer();
chandle = OpenFile("customer.log", &of, OF_READWRITE | OF_CREATE);
for(int i=0; i < citems; i++)
{
 Name = ( (BtreeEntry&) (BCustomer[i]) ). key();
 spos = ( (BtreeEntry&) (BCustomer[i]) ). value();
 _lwrite(chandle,Name,MAXNAMELEN);
 _lwrite(chandle,spos,POSLEN);
}
_lclose(chandle);


ditems = BDeposit.getItemsInContainer();
dhandle = OpenFile("deposit.log", &of, OF_READWRITE | OF_CREATE);
for(int j=0; j < ditems; j++)
{
 AccountNumber = ( (BtreeEntry&) (BDeposit[j]) ). key();
 spos = ( (BtreeEntry&) (BDeposit[j]) ). value();
 _lwrite(dhandle, AccountNumber, ACCOUNTLEN);
 _lwrite(dhandle,spos,POSLEN);
}
_lclose(dhandle);


bitems = BBranch.getItemsInContainer();
bhandle = OpenFile("branch.log", &of, OF_READWRITE | OF_CREATE);
for(int k=0; k < bitems; k++)
{
 Branch = ( (BtreeEntry&) (BBranch[k]) ). key();
 spos = ( (BtreeEntry&) (BBranch[k]) ). value();
 _lwrite(bhandle, Branch, MAXBLEN);
 _lwrite(bhandle,spos,POSLEN);
}
_lclose(bhandle);


titems = BBTransac.getItemsInContainer();
thandle = OpenFile("btransac.log", &of, OF_READWRITE | OF_CREATE);
for(int l=0; l < titems; l++)
{
 Branch = ( (BtreeEntry&) (BBTransac[l]) ). key();
 spos = ( (BtreeEntry&) (BBTransac[l]) ). value();
 _lwrite(thandle, Branch, MAXBLEN);
 _lwrite(thandle,spos,POSLEN);
}
_lclose(thandle);
```

```
 taitems = BATransac.getItemsInContainer();
 tahandle = OpenFile("atransac.log", &of, OF_WRITE | OF_CREATE);
 for(int m=0; m < taitems; m++)
 {
  AccountNumber = ( (BtreeEntry&) (BATransac[m]) ). key();
  spos = ( (BtreeEntry&) (BATransac[m]) ). value();
  _lwrite(tahandle, AccountNumber, ACCOUNTLEN);
  _lwrite(tahandle,spos,POSLEN);
 }
 _lclose(tahandle);


  delete &Name;
  delete &AccountNumber;
  delete &spos;
  delete &Branch;


 return TRUE;
}


/*
    The  destructor  frees  the  instance of  the  Borland's  custom

  contol  library  object  loaded  for  the  application
*/
TMainWindow::~TMainWindow()
{
 FreeLibrary((HINSTANCE) BWCCMod);
}


/*
     New  member function  executes  TNewDialog,  which  helps

  to open  a  new account for the  customer in  the  bank.
*/
void TMainWindow::New(RTMessage)
{
 GetModule()->ExecDialog( new TNewDialog(this,"NEWACCOUNT") );
}


/*
     NewBranch  member function  executes  TNewBranchDialog, which

  helps  to  open  a  new branch for the bank.
*/
void TMainWindow::NewBranch(RTMessage)
{
 GetModule()->ExecDialog(new TNewBranchDlg(this, "NEWBRANCH"));
}
```

```
/*
    Deposit member function executes TDepositDlg, which helps
  to deposit money to a customer's account.
*/
void TMainWindow::Deposit(RTMessage)
{
 GetModule()->ExecDialog(new TDepositDlg(this, "DEPOSIT"));
}

/*
    Withdraw member function executes TWithdrawDlg, which helps
  to withdraw money from a customer's account.
*/
void TMainWindow::Withdraw(RTMessage)
{
 GetModule()->ExecDialog(new TWithdrawDlg(this, "WITHDRAW"));
}

/*
    MoneyTransfer member function executes TTransferDlg, which
  helps to transfer money between two customer accounts.
*/
void TMainWindow::MoneyTransfer(RTMessage)
{
 GetModule()->ExecDialog(new TTransferDlg(this, "MONEYTRANSFER"));
}

/*
    ModifyAddress member function helps to modify a customer's
  address. ModifyAddress checks the given accountnumber for the
  presence in BDeposit Btree. If the given account number is valid,
  it executes TModifyDlg.
*/
void TMainWindow::ModifyAddress(RTMessage)
{
 long pos;
 int hHandle;
 OFSTRUCT of;

 char spos[POSLEN], AccountNumber[ACCOUNTLEN];
 char Name[MAXNAMELEN], SGlobalCustomerPos[POSLEN];

 sprintf(AccountNumber, "");

 PTInputDialog = new TInputDialog(this,"Account Number",
```

```
   " Enter the Account Number:", AccountNumber,sizeof AccountNumber);
  if( GetModule()->ExecDialog(PTInputDialog) == IDOK )

   {
    setstring(AccountNumber, ACCOUNTLEN);
    BtreeEntry AccountEntry(AccountNumber,"");
    int memberaccount = BDeposit.hasMember(AccountEntry);
    if(memberaccount)                    // if valid  account number
    {
    strcpy(spos, ( (BtreeEntry&) (BDeposit.findMember(AccountEntry))).value());
    pos = atol(spos);
    GlobalDepositPos = pos;
    hHandle = OpenFile("DEPOSIT.$$$", &of, OF_READ);
    _llseek(hHandle, (GlobalDepositPos + ACCOUNTLEN), 0);
    _lread(hHandle, Name, MAXNAMELEN);
    _lclose(hHandle);

    BtreeEntry NameEntry(Name, "");
    strcpy(SGlobalCustomerPos,((BtreeEntry&)
(BCustomer.findMember(NameEntry))).value());
    GlobalCustomerPos = atol(SGlobalCustomerPos);


    PTModifyDlg = new TModifyDlg(this, "MODIFYCUSTOMERINFO");
    GetApplication()->MakeWindow(PTModifyDlg);
    ShowWindow(PTModifyDlg->HWindow, SW_SHOW);  .

    }
    else                          // Not a  valid account Number
    {
     MessageBox(HWindow, "This Account Number does not exist", "Input Error", MB_OK);
    }
   }

}



/*
     ViewAccount member function  helps  to  view a customer's  account

   details:  his address,  his account number,  his balance , his  branch

   name and his transactions. ViewAccount checks the given  accountnumber

   for its presence in  BDeposit Btree. If  the  given account number is

   valid,  it   stores  the   latest twenty  transactions in  ListBoxData

   datamember and it executes TViewAccountDlg.
*/
void TMainWindow::ViewAccount(RTMessage)
{
  OFSTRUCT of;
  BOOL There;
  long pos, items, RealRank, AccountRank, nrank, itype;
```

```
int hHandle, tHandle, TrNumber = 1, flag = 1, loop1, i,j,k;
long Max, A[TRANDISPLAY], Value;
char spos[POSLEN], Type[TTYPELEN], SType[10], AccountNumber[ACCOUNTLEN];
char Name[MAXNAMELEN], SGlobalCustomerPos[POSLEN], SAmount[MAXDEPLEN];
int D, M, H, Mi, S;
char sd[3], sm[3], sy[5], sh[3], smi[3], ss[3], str[60];
char SBalance[MAXDEPLEN];

sprintf(AccountNumber, "");

PTInputDialog = new TInputDialog(this,"Account Number",
  " Enter the Account Number:", AccountNumber,sizeof AccountNumber);
if( GetModule()->ExecDialog(PTInputDialog) == IDOK )

 {
  setstring(AccountNumber, ACCOUNTLEN);
  BtreeEntry AccountEntry(AccountNumber,"");
  int memberaccount = BDeposit.hasMember(AccountEntry);
  if(memberaccount)
  {
  strcpy(spos, ( (BtreeEntry&) (BDeposit.findMember(AccountEntry))).value());
  pos = atol(spos);
  GlobalDepositPos = pos;



  hHandle = OpenFile("DEPOSIT.$$$", &of, OF_READ);
  _llseek(hHandle, (GlobalDepositPos + ACCOUNTLEN), 0);
  _lread(hHandle, Name, MAXNAMELEN);
  _lclose(hHandle);

  BtreeEntry NameEntry(Name, "");
  strcpy(SGlobalCustomerPos,((BtreeEntry&)
(BCustomer.findMember(NameEntry))).value());
  GlobalCustomerPos = atol(SGlobalCustomerPos);


  CustomerList = new TListBoxData();

  AccountRank = BATransac.rank((BtreeEntry&)AccountEntry);
  items = BATransac.getItemsInContainer();
  nrank = AccountRank;

  while ((flag) && ( nrank < (items-1)))
  {
   nrank = nrank + 1;
   flag = ((BtreeEntry&) (BATransac[nrank])).isEqual(AccountEntry);
   if(flag)TrNumber = TrNumber + 1;
  }

  flag = 1;
  nrank = AccountRank;
```

```
while((flag) && ( nrank > 0))
{
 nrank--;
 flag = ((BtreeEntry&) (BATransac[nrank])).isEqual(AccountEntry);
 if(flag)TrNumber = TrNumber + 1;
}

if(nrank>0)RealRank= nrank+1;
else RealRank=nrank;

if(TrNumber <= TRANDISPLAY)loop1 = TrNumber;
else loop1 = TRANDISPLAY;

 for(int l=0; l < TRANDISPLAY; l++) A[l] = -1;

 long p;
 for(i=0; i < loop1; i++)
 {
  Max = -5;
  for(j=0; j < TrNumber; j++)
  {
   k=0;
   p = RealRank + j;
   Value = atol(((BtreeEntry&)(BATransac[p])).value());
   There = FALSE;
   while((k < i) && (!There))
   {
       if(Value == A[k])There = TRUE;
       k++;
   }
   if(!There)
   {
       if(Value > Max)Max = Value;
   }
  }          // end of FOR loop
  A[i] = Max;
 }


tHandle = OpenFile("TRANSAC.$$$", &of, OF_READ);

for(int n=0 ; n< loop1 ; n++)
{
 _llseek(tHandle, A[n], 0);
 _llseek(tHandle, (ACCOUNTLEN + MAXBLEN), 1);
 _lread(tHandle, Type, TTYPELEN);
 _lread(tHandle, SAmount, MAXDEPLEN);
 _lread(tHandle, SBalance, MAXDEPLEN);
 _lread(tHandle, sd, DMHMISLEN);
 _lread(tHandle, sm, DMHMISLEN);
 _lread(tHandle, sy, YEARLEN);
 _lread(tHandle, sh, DMHMISLEN);
 _lread(tHandle, smi, DMHMISLEN);
 _lread(tHandle, ss, DMHMISLEN);
```

```
D = atoi(sd);
M = atoi(sm);
H = atoi(sh);
Mi= atoi(smi);
S = atoi(ss);


itype = atoi(Type);
switch (itype)
{
 case 1: strcpy(SType,"Open");
          setstring(SType,7);
          break;
 case 2: strcpy(SType,"Cr");
          setstring(SType,7);
          break;
 case 3: strcpy(SType,"Db");
          setstring(SType,7);
          break;
 case 4: strcpy(SType,"intr");
          setstring(SType,7);
          break;
 case 5: strcpy(SType,"Close");
          setstring(SType,7);
          break;
 case 6: strcpy(SType,"From");
          setstring(SType,7);
          break;
 case 7: strcpy(SType,"To");
          setstring(SType,7);
          break;
 default: ;
}

strcpy(str,"");
if(D < 10)
{
 strcat(str,"0");
 strncat(str, sd, 1);
}
else strcat(str, sd);
strcat(str, "/");
if(M < 10)
{
 strcat(str,"0");
 strncat(str, sm, 1);
}
else strcat(str, sm);
strcat(str, "/");
strcat(str, sy);
strcat(str,"    ");
```

```
    if(H < 10)
    {
     strcat(str,"0");
     strncat(str, sh, 1);
    }
    else strcat(str, sh);
    strcat(str, ":");
    if(Mi < 10)
    {
     strcat(str,"0");
     strncat(str, smi, 1);
    }
    else strcat(str, smi);
    strcat(str, ":");

    if(S < 10)
    {
     strcat(str,"0");
     strncat(str,ss,1);
    }
    else strcat(str, ss);
    strcat(str,"       ");
    strcat(str, SType);
    strcat(str,"      ");
    strcat(str, SAmount);
    strcat(str,"           ");
    strcat(str, SBalance);


    if(n == 0)CustomerList -> AddString(str, TRUE);
    else CustomerList -> AddString(str);

   }
   _lclose(tHandle);


   PTViewAccountDlg = new TViewAccountDlg(this, "VIEWACCOUNT");
   GetApplication()->MakeWindow(PTViewAccountDlg);
   ShowWindow(PTViewAccountDlg->HWindow, SW_SHOW);


   delete CustomerList;
   }
   else
   {
    MessageBox(HWindow, " Invalid Account Number Input", "Input Error", MB_OK);
   }
  }
}
```

```
   Branchdetails member function helps to view a given branch's
  details: branch name, branch city and branch assets. BranchDetails
  function checks the given branch name for the presence in
  BBranch Btree. If the given branch name is valid, it executes
  TBranchDetailsDlg.
*/
void TMainWindow::BranchDetails(RTMessage)
{
 long pos;
 char spos[POSLEN], Branch[MAXBLEN];

 sprintf(Branch, "");
 PTInputDialog = new TInputDialog(this,"Branch Name",
  " Enter the Branch Name:", Branch,sizeof Branch);
 if( GetModule()->ExecDialog(PTInputDialog) == IDOK )
 {
  setstring(Branch, MAXBLEN);
  BtreeEntry BranchEntry(Branch,"");
  int memberbranch = BBranch.hasMember(BranchEntry);
  if(memberbranch)
  {
  strcpy(spos, ( (BtreeEntry&) (BBranch.findMember(BranchEntry))).value());
  pos = atol(spos);
  GlobalBranchPos = pos;

  PTBranchDetailsDlg = new TBranchDetailsDlg(this, "BRANCHDETAILS");
  GetApplication()->MakeWindow(PTBranchDetailsDlg);
  ShowWindow(PTBranchDetailsDlg->HWindow, SW_SHOW);
  }
  else
  {
   MessageBox(HWindow, "Invalid Branch", "Input Error", MB_OK);
  }
 }
```

```
/*
        IsMonthEnd  member function returns  TRUE  on  the last day of

  the  month  after  6 pm  if that  month's interests for the accounts

  are not yet  calculated. Otherwise  it  returns  FALSE.
*/
BOOL TMainWindow::IsMonthEnd()
{
 OwnDate now, tomorrow;
 Time time;
 char sm[DMHMISLEN];
 int  handle, M, M1;
 OFSTRUCT of;
 M = now.Month();

 tomorrow = now + 1;


 if( tomorrow.Day() != 1)
 {
  MessageBox(HWindow, "Today is not the last day of the  month", "Input Error",MB_OK);
  return FALSE;
 };

 if(time.hour() < 18)            //  18 hours railway  time   or  6 pm
 {
  MessageBox(HWindow, "You can calculate interest only after 6 pm", "Input Error",MB_OK);
  return FALSE;
 };

 handle =  OpenFile("BANK", &of, OF_READ);
 _llseek(handle, ACCOUNTLEN, 0);
 _lread(handle, sm, DMHMISLEN);
 _lclose(handle);

 M1 = atoi(sm);
 if(M != M1)
 {
  MessageBox(HWindow, "Interest is already calculated for this month", "Input
Error",MB_OK);
  return FALSE;
 };

 if(M == 12)M = 1;
 else M++;

 itoa(M, sm, 10);
 setstring(sm, DMHMISLEN);
```

```
 handle =  OpenFile("BANK", &of, OF_WRITE);
 _llseek(handle, ACCOUNTLEN, 0);
 _lwrite(handle, sm, DMHMISLEN);
 _lclose(handle);

 return TRUE;
}


/*
   MonthlyUpdation member function helps calculating interest  for

   each account  of the savings bank for  the  given interest  on  the

   last day of the  month after 6 pm.
*/
void TMainWindow::MonthlyUpdation(RTMessage)
{
  if(IsMonthEnd())
  {
   sprintf(srate, "5.0");
   PTInterestDlg = new TInterestDlg(this,"Interest Rate",
    " Enter the Interest Rate:", srate,sizeof srate);
   GetModule()->ExecDialog(PTInterestDlg);
  }
}



/*
     CMExit  member function  closes  the  application.
*/
void TMainWindow::CMExit(RTMessage Msg)
{
  TWindowsObject::CMExit(Msg);
}


/*-------------------------------------------------*/
/*      TMainApp  implementations: */
/*-------------------------------------------------*/

/*
  Construct the TMainApp's MainWindow of type TMainWindow
*/
void TMainApp::InitMainWindow()
{
  MainWindow = new TMainWindow(NULL, "Savings  Bank");
}
```

```
/*
        InitInstance member function load  menu accelarators for  each

   instance  of  the  application.
*/
void TMainApp::InitInstance()
{
 TApplication::InitInstance();
 HAccTable = LoadAccelerators(hInstance, "SHORTCUTS");
}

//  The  application's  Main program
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpCmdLine, int nCmdShow)
{
 TMainApp MainApp("MainApplication", hInstance, hPrevInstance,
  lpCmdLine, nCmdShow);
 MainApp.Run();          // handles the message loop
 return MainApp.Status;   // returns  the  status of  the  application
}
```

6.2 Application's Header file (Project.h)

```
#include <owl.h>          // for the Object Window Library
#include <dialog.h>       // for TDialog class
#include <edit.h>         // for TEdit class
#include <bwcc.h>         // for Borland's custom control class library
#include <inputdia.h>     // for TInputDialog class
#include <listbox.h>      // for TListBox class
#include <stdio.h>        // for sprintf
#include <string.h>       // for strlen, strcpy and strcat
#include <stdlib.h>       // for atoi, atol and ltoa
#include <ctype.h>        // for isdigit and isalpha
#include <btree.h>        // for Btree class
#include <strng.h>        // for String class
#include <assoc.h>        // for association
#include <math.h>
#include <fcntl.h>        // for constants such as O_RDWR
#include <io.h>           // for open, read, write etc
#include <ltime.h>        // for Time class
#include "owndate.h"      // for OwnDate class
#include "wconst.h"       // for the IDs of the Dialog Boxes

#define MAXNAMELEN 32     // Maximum Customer Name string length
#define MAXSTRLEN 20      // Maximum Street Name string length
#define MAXCITYLEN 20     // Maximum City Name string length
#define PINLEN 8          // Pin Code string length 6
#define ACCOUNTLEN 7      // Account Number string length 5
#define MAXBLEN 20        // Maximum Branch Name string length
#define MAXDEPLEN 12      // Maximum Deposit possible 10 lakh Rs.
#define MAXWDLEN 12       // Maximum Withdraw possible 10 lakh Rs.
#define MAXBCLEN 20       // Maximum BranchCity string length
#define MAXBALEN 15       // Maximum Assets possible 100 Crore Rs.
#define DMHMISLEN 3       // string length for day, month, hour, min & sec
#define YEARLEN 5         // string length for year
#define TTYPELEN 2        // string length for Transaction type
#define CRECORDLEN 80     // CUSTOMER.$$$ file's record length
#define DRECORDLEN 83     // DEPOSIT.$$$ file's record length
#define BRECORDLEN 55     // BRANCH.$$$ file's record length
#define TRECORDLEN 73     // TRANSAC.$$$ file's record length
#define POSLEN 12         // File position's string length
#define TRANDISPLAY 20    // No. of transactions displayed for customer view
#define APPNAME "BANK"


void setstring(char *s, int c);
```

```
/*
  BtreeEntry Class is for storing  two related quatities( Key, Value).
  Given the Key you can have the Value corresponding to that Key. We are
  using this association with Btree structure for storing and accessing
  a key and that Key's  position in a file.
*/
class BtreeEntry: public Sortable
{

  String& aKey;
  String& aValue;

public:

  enum {BtreeEntryClass = __firstUserClass};

  BtreeEntry(char* name,char* value)   // BtreeEntry class's constructor
  : aKey(*new String(name) ),
    aValue(*new String(value) ) {}

  BtreeEntry()                    // BtreeEntry  class's constructor
  : aKey(*new String("")),
    aValue(*new String("") ) {}

  classType isA() const
  { return BtreeEntryClass; }

  char _FAR *nameOf() const
  { return "BtreeEntry"; }

  /*
     checks if  two  BtreeEntry  objects  are  equal based  on the key
  */
  int isEqual(const Object& e) const
  { return key() == ((BtreeEntry&)(Association&)e).key(); }

  int isLessThan(const Object& e) const
  {  return key() < ((BtreeEntry&)(Association&)e).key(); }

  hashValueType hashValue() const
  { return aKey.hashValue(); }

  void printOn(ostream& os) const
  { os << aKey << ":" << aValue; }

  String& key() const {return aKey;}

  String& value() {return aValue;}

  ~BtreeEntry() {delete &aKey; delete &aValue;}
};
```

```
/*
TModifyDlg class is for a dialog for modifying the address of a customer.
Declaring TModifyDlg as a descendant of TDialog .
*/
class TModifyDlg : public TDialog {
public:
   char Name[MAXNAMELEN];
   char Street[MAXSTRLEN];
   char City[MAXCITYLEN];
   char Pin[PINLEN];
   char NewName[MAXNAMELEN];
   PTEdit  PNameEdit, PStreetEdit, PCityEdit, PPinEdit;
   TModifyDlg(PTWindowsObject AParent, LPSTR name);
   void WMInitDialog(RTMessage Msg)= [WM_FIRST + WM_INITDIALOG];
   virtual BOOL CanClose();

private:
   void FillBuffers();
   BOOL IsAddressModified();
   BOOL IsNameModified();
   BOOL ValidName(char*);
   BOOL ValidStreet(char*);
   BOOL ValidCity(char*);
   BOOL ValidPin(char*);

};

/*
TNewDialog class is for a dialog for the customer to open a new bank account.
Declaring TNewDialog as a descendant of TDialog .
*/
class TNewDialog : public TDialog {
public:
   char Name[MAXNAMELEN];
   char Street[MAXSTRLEN];
   char City[MAXCITYLEN];
   char Pin[PINLEN];
   char BranchName[MAXBLEN];
   char Deposit[MAXDEPLEN];
   char AccountNumber[ACCOUNTLEN];
   TNewDialog(PTWindowsObject AParent, LPSTR name);
   void WMInitDialog(RTMessage Msg)= [WM_FIRST + WM_INITDIALOG];
   virtual BOOL CanClose();
private:
   void FillBuffers();
   BOOL ValidName(char*);
   BOOL ValidStreet(char*);
   BOOL ValidCity(char*);
   BOOL ValidPin(char*);
   BOOL ValidBranch(char*);
   BOOL ValidDeposit(char*);
};
```

```
/*
TNewBranchDlg class is for a dialog for the bank to open a new branch.
Declaring TNewBranchDialog as a descendant of TDialog .
*/
class TNewBranchDlg : public TDialog {
public:
   char BranchName[MAXBLEN];
   char BranchCity[MAXBCLEN];
   char BranchAsset[MAXBALEN];
   TNewBranchDlg(PTWindowsObject AParent, LPSTR name);
   virtual BOOL CanClose();
private:
   void FillBuffers();
   BOOL ValidBranchName(char*);
   BOOL ValidBranchCity(char*);
};


/*
TBranchDetailsDlg class is for a dialog for viewing Branch Details.
Declaring TBranchDetailsDlg as a descendant of TDialog .
*/
class TBranchDetailsDlg : public TDialog {
public:
   char BranchName[MAXBLEN];
   char BranchCity[MAXBCLEN];
   char BranchAsset[MAXBALEN];
   TBranchDetailsDlg(PTWindowsObject AParent, LPSTR name);
   void WMInitDialog(RTMessage Msg) = [WM_FIRST + WM_INITDIALOG];
};



/*
   TDepositDlg class is for a dialog for the customer to deposit money
  into his account.
  Declaring TDepositDlg as a descendent of TDialog
*/
class TDepositDlg : public TDialog {
public:
   char AccountNumber[ACCOUNTLEN];
   char Deposit[MAXDEPLEN];
   char Branch[MAXBLEN];
   TDepositDlg(PTWindowsObject AParent, LPSTR name);
   virtual BOOL CanClose();
protected:
   void FillBuffers();
   BOOL ValidAccount(char*);
   BOOL ValidDeposit(char*);
};
```

```
/*
   TWithdrawDlg class is for a dialog for the customer to withdraw money
  from his account.
  Declaring TWithdrawDlg as a descendent of TDialog
*/
class TWithdrawDlg : public TDialog {
public:
   char AccountNumber[ACCOUNTLEN];
   char Withdraw[MAXWDLEN];
   char Branch[MAXBLEN];
   TWithdrawDlg(PTWindowsObject AParent, LPSTR name);
   virtual BOOL CanClose();
protected:
   void FillBuffers();
   BOOL ValidAccount(char*);
   BOOL ValidWithdraw(char*);
};



/*
   TTransferDlg class is for a dialog for the customer to deposit money
  into his account.
  Declaring TTransferDlg as a descendent of TDialog
*/
class TTransferDlg : public TDialog {
public:
   char FromAccountNumber[ACCOUNTLEN];
   char ToAccountNumber[ACCOUNTLEN];
   char Deposit[MAXDEPLEN];
   TTransferDlg(PTWindowsObject AParent, LPSTR name);
   virtual BOOL CanClose();
protected:
   void FillBuffers();
   BOOL ValidAccount(char*);
   BOOL ValidMoneyTransfer(char*);
};


/*
TViewAccountDlg class is for a dialog for viewing Customer Details.
Declaring TViewAccountDlg as a descendant of TDialog .
*/
class TViewAccountDlg  : public TDialog {
public:
   char Name[MAXNAMELEN];
   char Street[MAXSTRLEN];
   char City[MAXCITYLEN];
   char Pin[PINLEN];
   char AccountNumber[ACCOUNTLEN];
   char BranchName[MAXBLEN];
   char Balance[MAXDEPLEN];
   PTListBox CustomerListBox;
   PTStatic  PName, PStreet, PCity, PPin, PAccount, PBranch, PBalance;
   TViewAccountDlg(PTWindowsObject AParent, LPSTR name);
   void WMInitDialog(RTMessage Msg) = [WM_FIRST + WM_INITDIALOG];
```

```
};

class TInterestDlg  :  public TInputDialog {
public:
   char SRate[6];
   TInterestDlg(PTWindowsObject, LPSTR, LPSTR, LPSTR, WORD);
   BOOL ValidRate(char*);
   BOOL CanClose();
};

/* Declare TMainWindow, a TWindow descendant */
class TMainWindow : public TWindow {
public:
   HANDLE BWCCMod;
   TModifyDlg *PTModifyDlg;
   TBranchDetailsDlg *PTBranchDetailsDlg;
   TInputDialog *PTInputDialog;
   TInterestDlg *PTInterestDlg;
   TViewAccountDlg *PTViewAccountDlg;
   PTListBoxData CustomerList;

   TMainWindow(PTWindowsObject AParent, LPSTR ATitle);
   ~TMainWindow();
   LPSTR  GetClassName();
   virtual void GetWindowClass(WNDCLASS&);
   virtual void New(RTMessage Msg) = [CM_FIRST + CM_NEW];
   virtual void NewBranch(RTMessage Msg) = [CM_FIRST + CM_NEWBRANCH];
   virtual void Deposit(RTMessage Msg) = [CM_FIRST + CM_DEPOSIT];
   virtual void Withdraw(RTMessage Msg) = [CM_FIRST + CM_WITHDRAW];
   virtual void ModifyAddress(RTMessage Msg) = [CM_FIRST + CM_MODIFY];
   virtual void MoneyTransfer(RTMessage Msg) = [CM_FIRST + CM_TRANSFER];
   virtual void BranchDetails(RTMessage Msg) = [CM_FIRST + CM_VIEWBRANCH];
   virtual void ViewAccount(RTMessage Msg) = [CM_FIRST + CM_VIEW];
   virtual void MonthlyUpdation(RTMessage Msg) = [CM_FIRST + CM_INTEREST];
   virtual void CMExit(RTMessage Msg) = [CM_FIRST + CM_EXIT];
   virtual BOOL IsMonthEnd();
   virtual BOOL CanClose();
};

/* Declare TMainApp, a TApplication descendant */
class TMainApp : public TApplication {
public:
  TMainApp(LPSTR name, HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
    : TApplication(name, hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
  virtual void InitInstance();
};
```

6.3 Application's Module definition file (standard.def)

```
NAME    BankApp
DESCRIPTION    "Savings Bank"
STUB  "WINSTUB.EXE"   //   Gives message  " Program Runs under  Windows only "
                      //   if  you  run  it  outside the  Windows  environment
EXETYPE WINDOWS
CODE PRELOAD MOVEABLE DISCARDABLE   //  For the  Windows  to  manage
DATA PRELOAD MOVEABLE MULTIPLE      //  memory  of  the application
HEAPSIZE 8192
STACKSIZE 5120
```

Bibliography

1. "ObjectWindows for C++ User's Guide", Borland International Inc.
2. "Microsoft Windows Software Development Kit Guide to Programming", Microsoft Corporation.
3. Alan Synder, "The essence of Objects: Concepts and Terms", IEEE Software January, 1993.
4. "A Structured approach to Object - Oriented Design", Addendum to the proceedings OOPSLA'91, pp 21-43.
5. Bjarne Stroustroup, "The C++ Programming Languge", Addison Wesley Publishing Company, 1991.
6. Brian W. Kernighan and Dennis M. Ritchie, "The C Programming Language", Printice Hall of India Private Limited, 1991
7. Gary Syck, "ObjectWindows How - To", Galgotia Publications Private Limited, 1993.
8. Henry F. Korth and Abraham Silberschatz, "Database System Concepts", McGraw-Hill Computer Science Series, 1991.
9. James Conger, "Windows API Bible", Galgotia Publications Private Limited, 1993.
10. Jim Conger, "Windows Programming Primer Plus", Galgotia Publications Private Limited, 1993.
11. Louis Fernandes, Yogesh Sheth and Anish Kurup, "Borland C++ 3.0 for Windows 3.1 Programming with ObjectWindows", BPB Publications, 1993.
12. Nambiar K. K., "Some Analytic Tools of Database Design of Relational Database Systems", Proc. 6th Conference on Very Large Databases, Montreal(1980), IEEE 1980.

13. Pankaj Jalote, "An Integrated Approach to Software Engineering", Narosa Publishing House, 1991.

14. Peter Wegner, "Dimensions of Object - Oriented Modelling", IEEE Computer October 1992, pp 12 - 20.

15. Robert Lafore, "Object Oriented Programming in Turbo C++ ", Galgotia Publications Private Limited, 1992.

16. Setrag Khoshafian and Razmik Abnous, "Object Orientation Concepts, Language, Database, User Interfaces", John Wiley & Sons Inc., 1990.

17. Stanley B. Lippman, "C++ Primer", Addison Wesley Publishing Company, 1991.

18. Stevens A. L., "C Database Development", BPB Publications, 1989.

Glossary

Abstraction: Abstraction is a tool that permits the designer to consider a component' s external behaviour without worrying about its internal details.

Object: Object is an abstraction of a real world entity. The objects can be a dialog, an aeroplane etc for example.

Class: Class is a collection of objects

Data member: Variables in a class are called data members.Synonym for data member is instance variable.

Method: Procedures defined for a class are called methods. synonym for method is member function.

Base class: Base class is a class from which new classes are derived.

Derived class: Derived class is a class derived from a base class.

Inheritance: Inheritance is a concept which allows an object to inherit the attributes of another object.

Muliple Inheritance: Multiple Inheritance is Inheritance of attributes from multiple objects.

Encapsulation: Encapsulation is hiding of internals of an object. Synonyms for Encapsulation are data hiding and information hiding.

Window: When the screen is split into several independent regions, each region is called a window.

Icon: Icon is a small graphic image that represents an application when that application's main window is minimized.

Menu: A menu is a list commands that the user can view and choose from.

Mouse: A mouse is a pointing device, which allows users to point at different parts of the screen.

Button: A button is a small labelled window and it cause an action to take place when the user selects it.

Dialog box: Dialog box is a window that application uses to interact with the user.

List box: List box is a window which is used to display a list of strings and one

or more strings can be selected from the window.

Edit contol: Edit control is a window which allows users to edit, display or modify text.

Static control: Static contol is a window which allows users to display text.

Accelarator: Accelarator is a keyboard shortcut to a menu command.

Resource: Resource is a read only data stored in an application's executable file that Windows reads from disk on command. They are used to manage windows and user - defined objects like menus, bitmaps, dialog boxes, icons and accelarators.